

# Distributed Agent-Based Ant Colony Optimization for Solving Traveling Salesman Problem on a Partitioned Map

Sorin Ilie  
University of Craiova  
Software Engineering  
Department  
Bvd.Decebal 107, Craiova,  
200440, Romania  
sorin.ilie@software.ucv.ro

Amelia Bădică  
University of Craiova  
Business Information Systems  
Department  
Str.A.I.Cuza 13, Craiova,  
200585, Romania  
ameliabd@yahoo.com

Costin Bădică  
University of Craiova  
Software Engineering  
Department  
Bvd.Decebal 107, Craiova,  
200440, Romania  
costin.badica@software.ucv.ro

## ABSTRACT

In this paper we discuss the experimental evaluation of an improved configuration of our recent framework ACODA (Ant Colony Optimization on a Distributed Architecture) for solving the Traveling Salesman Problem (TSP). ACODA is a novel multi-agent system architecture for distributed Ant Colony Optimization in a decentralized environment. This new configuration improves the execution time by allowing each software agent of ACODA to manage a part of the TSP map rather than a single map node. Experimental results support this claim of improved efficiency while showing that the scalability of the system is preserved.

## Categories and Subject Descriptors

I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence; D.2.11 [Software]: Software Architectures

## General Terms

Algorithms, Design, Experimentation

## 1. INTRODUCTION

Natural bio-systems are composed of many natural entities engaged in complex behaviors, interactions and processes according to the laws of nature. There is a certain level of abstraction at which such systems can be modeled as distributed computational systems composed of interacting artificial entities. Thus, we would expect distributed computing to have a lot of potential for the application of nature-inspired computing – i.e. computing inspired by behaviors of natural systems.

In particular, Ant Colony Optimization (ACO hereafter) is inspired by collective behavior of colonies of natural ants when they explore the environment searching for food. In this context, we support the idea that ACO should allow a straightforward mapping onto existing distributed architec-

tures, including multi-agent systems middleware. Therefore, to take advantage of the full potential of nature inspired computational approaches, we have started investigation of new distributed forms of ACO using state-of-the-art multi-agent technology. The focus of our research is to propose a better computational architecture for implementation of ACO algorithms based on distributed multi-agent middleware. We foresee two potential benefits of our work: (i) first, it contributes to the engineering of ACO algorithms using distributed computing architectures, thus resulting in more scalable and efficient algorithms; (ii) second, as a byproduct, our work might contribute to better understanding of new forms of agent-based distributed ACO.

Based on our literature review we have found that most of the works related to distributed ACO do not address the distribution problem as we do. Rather, existing approaches rely on mapping the classical sequential algorithms onto parallel and high-performance computing architectures without a careful and in-depth consideration of the intrinsic distributed nature of ACO.

Standard work [6] discusses only sequential implementations of various ACO approaches. Basically this means that a single sequential program manages the map, the ants, the ant movements and the pheromone deposits. Attempts at a fine grained parallelization of ACO [12] abandon this approach in favor of separating the ants from the environment. However this proposal introduces additional communication overhead between the ants themselves and between the ants and the environment. Other approaches [3, 5, 13] set up the goal of coarse grained parallelization starting from a sequential ACO algorithm like those proposed in [6]. The algorithm is run in parallel on multiple machines, simultaneously on the same map. Note however that with this approach, the different instances of ACO will need to synchronize after each computation cycle that yields a potential solution, and in our opinion this synchronization limits the potential of the fully asynchronously distributed behavior that can be observed in natural ants. Our approach starts from the initial algorithms of [6] and proposes a new distributed architecture [9, 10] for ACO that better simulates the distributed behavior of colonies of natural ants that live in natural environments.

In our recent work [9] we proposed a scalable multi-agent system architecture called ACODA (Ant Colony Optimization on a Distributed Architecture) that allows the implementation of ACO in a parallel, asynchronous and decentral-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WIMS'11 May 25-27, 2011 Sogndal, Norway

Copyright 2011 ACM 978-1-4503-0148-0/11/05 ...\$10.00.

ized environment. The novelty of our approach is: (i) the problem environment is conceptualized and implemented as a distributed multi-agent system ([14, 4]) and (ii) ant management is reduced to messages exchanged asynchronously between the agents of the problem environment. The initial scalability of ACODA was assessed by evaluating its application to Traveling Salesman Problem (TSP hereafter) on a high-speed cluster network of 7 ordinary computers.

There are two main informal arguments that motivate the suitability of our approach for the distributed implementation of ACO algorithms. The first motivates the use of multi-agent systems for modeling the ants' environment. The second motivates the mapping of the ants' environment to a distributed architecture and the mapping of the ants' migration to messages exchanged between the agents located in the ants' environment.

First, although behavior of natural ants can be described as complex, this complexity is mainly a consequence of the complexity of the physical environment in which the ants live and act, rather than of the ants' intrinsic "intelligence". This can be noticed by carefully observing that ants movement in their environment actually "mimics" the complexity of this environment, and therefore their sequence of decisions and movements in searching for a path appears to be very complex and "intelligent" to an external observer. This observation supports the idea of multi-agent modeling of the ants' environment as a complex system, while on the other hand it suggests keeping the ants as simple as messages exchanged between the agents.

For the second motivation let us consider an ACO problem which is represented as a multi-agent system consisting of  $n$  agents. Intuitively, by distributing the agents on several machines, the agent communication load is divided between them thus resulting in the execution time being reduced. To simplify our analysis we assume that there is the same number  $N$  of ant migrations between any two agents in the system. We further assume that the cost of exchanging an ant message on a single machine is equal to constant  $a$ , while the cost of exchanging an ant message between two different machines is  $b > a$ . If the system is mapped to a single machine it follows that the execution time is at least  $a * N * n * (n - 1)/2$ , i.e. the total cost required to exchange all the ant messages. However, if we map each agent to a separate machine then communication cost of each machine is  $b * N * (n - 1)$ . Now, if  $n > 2 * b/a$ , i.e. if the number of machines is sufficiently large, then the execution time of each machine will be reduced because its communication load is reduced.

The focus of this paper is to improve the approach introduced in [9] for the application of ACODA to TSP. The goal of TSP is to compute a shortest tour that visits exactly once each node of a map modeled as a complete weighted graph. Briefly (more details are given in subsequent sections of this paper), in our initial approach we configured ACODA with one agent per map node. While this approach allows the distribution of the agents on the available machines, it clearly has the drawback of using too many agents, i.e. a high communication overhead in the underlying agent platform. For example, for a map of 1000 nodes, 1000 agents must be created, which could be a problem if, for example, only a network of few (let's say 7) ordinary computers is available. On average, we shall have 142 (i.e. approximately  $1000/7$ ) agents per network machine. Taking into account that on

an agent platform like JADE (that we used in our experiments, see [4]) an agent consumes one thread, this could be a serious limitation of the approach – i.e. what you gain by distributing computation on many machines, you actually lose by the higher overhead of running too many threads on each single machine.

So in this paper we propose an improved version of our approach [9] by allowing each software agent which is part of an ACODA configuration to manage more than one map node – i.e. a part of the TSP map. So, the initial TSP map is partitioned into a set of disjoint sub-maps and each sub-map is managed by a single different ACODA agent. The effect of this approach is: (i) the total number of agents created for an ACODA configuration for solving a given TSP problem is reduced times the size (i.e. number of nodes) of a sub-map; and (ii) this in turn reduces significantly the communication overhead, as well as the number of threads on each machine of the ACODA configuration. Note that our approach does not attempt to solve each sub-map separately and then combine the sub-solutions into a solution, but rather we attempt to directly solve the whole TSP map using the distributed partitioned approach.

The paper is structured as follows. We start in section 2 with a background on ACO algorithms. We follow in section 3 with a detailed presentation of our approach of mapping ACO algorithms to a distributed multi-agent architecture based on JADE framework and implicitly the current configuration of ACODA. In section 4.1 we present experimental results obtained by applying our framework to TSP problem. In section 5 we briefly compare our solution with existing solutions proposed in the literature.

## 2. BACKGROUND

ACO refers to a family of optimization algorithms that get their inspiration from the metaphor of real ants searching for food. When natural ants are searching for new food sources, they secrete pheromone on their way back to the anthill. Other ants of the colony may sense the pheromone and become attracted by the marked paths; the more pheromone is deposited on a path, the more attractive that path becomes. The pheromone is volatile, so it disappears over time. Evaporation erases pheromone on longer paths as well as on paths that are not of interest anymore. However, shorter paths are more quickly refreshed, thus having the chance of being more frequently explored. Intuitively, ants will converge towards the most efficient path, as that path gets the strongest concentration of pheromone. Artificial ants are programmed to mimic this behavior of real ants while searching for food.

ACODA is a distributed architecture specially designed to support efficient implementation of distributed ACO algorithms. ACODA was already applied for solving TSP ([9]). The goal of TSP is to compute a shortest tour that visits each node of a complete weighted graph exactly once. The decision version of TSP is NP-complete which basically means that it is very unlikely that a polynomial solution for solving TSP exists. So TSP is a very good candidate for the application of heuristic approaches, including ACO.

The main idea behind ACODA is to provide an agent-based distributed infrastructure for modeling the problem environment ([14, 4]). Artificial ants originating from the anthills that are located in the environment will travel to find optimal solutions, following ACO rules. In order to use ACO to solve TSP, the problem environment is concep-

tualized as a distributed set of interconnected graph nodes that are managed by software agents (note however that this modeling is specific to TSP, so it must be updated when ACODA is applied to solving other problems). Ants travel between nodes following ACO rules until they find a solution, i.e. until they complete a tour for TSP. Once they return to their originating anthill, they mark the solution with pheromone by tracing back their path. In ACODA, ant traveling is realized by messages exchanged asynchronously between the agents located in the problem environment. In [9] we have used a single node per agent, while in this paper we investigate the effect of allocating more nodes to a single agent.

We used in our experiments the distributed version of the ACS algorithm [6], initially proposed in [9]. The original ACS is a sequential implementation of ACO which chose to interleave the movement of ants rather than separately moving each ant until it finishes its tour. Our approach presents a few differences due to the requirements of a truly distributed architecture based on asynchronous message passing and the avoidance of using global knowledge, this fitting to the basic principles of ACODA.

In what follows we briefly review the mathematical model of ACO that we employed as the basis for our system design and implementation.

ACO rules determine the amount of pheromone deposited on edges, the edge chosen by each ant on its way, and how fast the pheromone deposited on each edge evaporates. For this purpose we use the mathematical model of ACO that was proposed in ACS. Ant  $a$  located at node  $i$  decides to move to node  $j$  with the probability  $p_{i,j}$  computed as follows:

$$p_{i,j} = \frac{(\tau_{i,j})^\alpha (\eta_{i,j})^\beta}{\sum_j (\tau_{i,j})^\alpha (\eta_{i,j})^\beta} \quad (1)$$

where:

- $\tau_{i,j}$  is the amount of 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  $\tau_{i,j}$
- $\beta$  is a parameter to control the influence of  $\eta_{i,j}$
- $j$  represents a node reachable from node  $i$  that was not visited yet

Better solutions need to be marked with more pheromone. So whenever an ant  $a$  determines a new tour  $V_a$  of cost  $L_a$ , the ant will increase pheromone strength on each edge of the tour with a value that is inversely proportional to the cost of the tour.

$$\Delta\tau_{i,j}^a = \begin{cases} 1/L_a & \text{if edge } (i, j) \text{ belongs to found tour } V_a \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where  $L_a$  is the cost of the  $a$ -th ant's tour.

When an ant travels along a given path, this traveling takes an amount of time that is proportional with the travel distance (assuming the ants move with constant speed). As pheromone is volatile, if a real ant travels more, pheromone will have more time to evaporate, thus favoring better solutions to be discovered in the future. We can conclude that

adding pheromone evaporation to our model can be useful, especially for solving a complex problem like TSP.

When an ant completes a tour it will trace back its steps marking the edges on the way with pheromone. The update will also take into account pheromone evaporation. Both evaporation and pheromone updates are implemented in ACS as follows:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \rho\Delta\tau_{i,j}^a \quad (3)$$

where  $\rho$  is the evaporation rate  $0 \leq \rho < 1$ .

All ants use equation 1 to probabilistically determine their next step. Therefore they will often choose the edge with the highest pheromone concentration, i.e. the exploration of less probable edges is low. The solution to increase the exploration of alternative paths is to remove some pheromone on edges chosen by ants, i.e. apply a local evaporation process. This has the effect of making them less desirable, increasing the exploration of the edges that have not been picked yet. So whenever an ant travels along an edge, it applies local evaporation by updating pheromone as follows:

$$\tau_{i,j} = (1 - \xi)\tau_{i,j} + \xi\tau_0 \quad (4)$$

where:

- $\xi$  is the local evaporation rate  $0 \leq \xi < 1$ .
- $\tau_0$  is the initial amount of pheromone on each edge

A good heuristics to initialize pheromone trails is to set them to a value slightly higher than the expected amount of pheromone deposited by the ants in one tour; a rough estimate of this value can be obtained by setting  $\tau_0 = 1/(nC)$ , where  $n$  is the number of nodes, and  $C$  is the tour cost generated by a reasonable tour approximation procedure [6]. For example we can set  $C = nw_{avg}$  where  $w_{avg}$  is the average edge cost.

### 3. ARCHITECTURE

In ACODA, the problem environment (i.e. the TSP map in our case) is designed as a set of interconnected software agents [14]. For the purpose of this work, by software agent (agent in what follows) we understand a software entity that: (i) has its own thread of control and can decide autonomously if and when to perform a given action; (ii) communicates with other agents by asynchronous message passing. Each agent is referenced using its name, also known as agent identifier.

The activity carried out by a given agent is represented as a set of behaviors. A behavior is defined as a sequence of primitive actions. Behaviors are executed in parallel using interleaving of actions on the agent's thread with the help of a non-preemptive scheduler, internal to the agent [4].

Here we propose a configuration of ACODA that allows agents to manage a sub-map of the initial TSP map. Using this approach, ants can move between local nodes (i.e. nodes managed by the same agent) and between nodes that are managed by different agents. In the second case, an ant move assumes the exchange of a message flowing from the agent that manages the source node to the agent that manages the destination node.

Agent design must include behaviors for moving ants in both situations. Whenever an agent receives an ant, its

RECEIVE-ANT() behavior (see table 1) immediately updates it and then sends it out to a neighbor node following ACO rules. At this point, ants that must move to local nodes will be pushed onto the local ant queue (there is one such queue for each agent), while the other ants are sent out as messages to the agent that manages the destination node. A second behavior, LOCAL-MOVE(), continually gets the next ant from the local ant queue and sends it out to a neighbor node following ACO rules.

Ants are represented as objects with a set of attributes: cost of the currently search path (which becomes  $L_a$  when the ant completes a tour), pheromone strength (value of  $\Delta\tau_{i,j}^a$ ), returning flag, best tour cost (value of the currently best tour that the ant knows, based on its search history) and a list of node identifiers representing the path that the ant followed to reach its current location. The list is necessary for two reasons: i) the ant needs to trace back its steps in order to mark the tour with pheromone and ii) we need to avoid loops so only unvisited nodes are taken into account as possible next hops. Attributes are initialized when an ant is created. Subsequently, they are updated during the process of ant migration to reflect the current knowledge of the ant about the explored environment. For example, whenever an ant completes a tour by reaching its destination node (i.e. its anthill), the agent that is managing that node saves locally the ant's currently best tour. Whenever another ant from the colony travels through this node, it is able to sense the environment and eventually to update its "current knowledge" about the value of the best tour found so far. So ants have the possibility to exchange via the environment (i.e. the set of agents that are managing the map nodes) not only pheromone information, but also information about the best solutions found so far.

Each agent has the following parameters: value of  $\rho$ , value of  $\xi$ , best tour cost (the value of the currently best tour that the agent knows, based on ants that traveled through this agent), a set of lists of neighbor nodes (a list for each node managed by this agent) including the weights and the values of deposited pheromone of the corresponding edges, and a hash map that associates to each node the identifier of its managing agent. Parameters  $\rho$  and  $\xi$  are utilized to implement equations 3 and 4. Note that each agent and each ant maintain their own values of the currently best tours they encountered so far. So, whenever an ant is reaching a node, the ant and the agent managing that node are able to exchange and update their best tour information accordingly.

In our approach, each agent creates its own ant population for each of the nodes that it manages. The agents calculate pheromone strength according to the tour cost (see equation 2) and also update the ants' pheromone strength attribute. Additionally, the agents set the returning flag for returning ants, exchange ant information with other agents, deposit pheromone when needed, and update the cost of the current search path of an ant by adding the weight  $w$  of the edge that was chosen using equation 1.

The functionality of an agent is as follows. RECEIVE-ANT() behavior retrieves an ant message from the agent's message queue (if this queue is not empty, see [4]), then parses the ant message, adjusts the ant's attributes using ADJUST-ATTRIBUTES() method and sends it to the node determined by BEST-NEIGHBOR() method (this could be either a local node or a node external to this agent). ADJUST-ATTRIBUTES() method sets returning flag and calculates

pheromone strength using equation (2) whenever an ant has completed a tour. Ants that have returned to the anthill are re-initialized.

BEST-NEIGHBOR() method uses equation (1) to determine the address of the node where to send the ant. When the ant returns to the anthill, this method sends the ant to the first node from its list of visited nodes, popping it from the list, and deposits the ant's pheromone on the corresponding edge. This method also implements evaporation using equations 4 and 3. The SEND-TO() method either sends out an ant as a message to the agent that manages the destination node or pushes the ant onto the agent's local ant queue if the destination node is managed by the current agent. The LOCAL-MOVE() behavior pops an ant object from the local ant queue, adjusts the ant's attributes using ADJUST-ATTRIBUTES() method and sends it out to the node determined by BEST-NEIGHBOR() method.

By allowing each ACODA agent to manage more nodes, we aim to decrease the total execution time by using local message exchanges whenever possible. The intuition behind

**Table 1: Algorithms for processing ant information.**

|                                                                            |
|----------------------------------------------------------------------------|
| RECEIVE-ANT()                                                              |
| 1. RECEIVE( <i>ant</i> )                                                   |
| 2. ADJUST-ATTRIBUTES( <i>ant</i> )                                         |
| 3. SEND-TO( <i>ant</i> ,BEST-NEIGHBOR( <i>ant</i> ))                       |
| LOCAL-MOVE()                                                               |
| 1. DEQUEUE( <i>ant</i> )                                                   |
| 2. ADJUST-ATTRIBUTES( <i>ant</i> )                                         |
| 3. SEND-TO( <i>ant</i> ,BEST-NEIGHBOR( <i>ant</i> ))                       |
| ADJUST-ATTRIBUTES( <i>ant</i> )                                            |
| 1. <b>if</b> AT-ANTHILL( <i>ant</i> ) <b>then</b>                          |
| 2. <b>if</b> RETURNING( <i>ant</i> ) <b>then</b>                           |
| 3.     INITIALIZE( <i>ant</i> )                                            |
| 4. <b>else</b>                                                             |
| 5.     SET-RETURN-FLAG( <i>ant</i> )                                       |
| 6. $\triangleright$ calculate ant pheromone using equation 2.              |
| 7.     CALCULATE-PHEROMONE-STRENGTH( <i>ant</i> )                          |
| 8.     UPDATE-BEST-TOUR()                                                  |
| BEST-NEIGHBOR( <i>ant</i> )                                                |
| 1. <b>if</b> RETURNING( <i>ant</i> ) <b>then</b>                           |
| 2. $\triangleright$ update pheromone using equation 3.                     |
| 3.   DEPOSIT-PHEROMONE()                                                   |
| 4. <b>return</b> LAST-VISITED-NODE( <i>ant</i> )                           |
| 5. $\triangleright$ ant performs random choice according to equation 1.    |
| 6. $bestNeighbor \leftarrow \text{RANDOM-CHOICE}()$                        |
| 7.   UPDATE-CURRENT-PATH-COST( <i>bestNeighbor</i> )                       |
| 8. $\triangleright$ apply local evaporation of pheromone using equation 4. |
| 9.   LOCAL-EVAPORATE-PHEROMONE()                                           |
| 10.   ADD-TO-VISITED-LIST( <i>bestNeighbor</i> , <i>ant</i> )              |
| 11. <b>return</b> <i>bestNeighbor</i>                                      |
| SEND-TO( <i>ant</i> , <i>destination</i> )                                 |
| 1. <b>if</b> LOCAL( <i>destination</i> ) <b>then</b>                       |
| 2.   ENQUEUE( <i>ant</i> )                                                 |
| 3. <b>else</b>                                                             |
| 4.   SEND( <i>ant</i> , <i>destination</i> )                               |

this is that a local ant move is significantly faster than an ant move between two different agents. An important consequence of this statement is that the execution time will depend on the map partition. With the current approach this is fixed and configured before the setup of the system. However, this issue of the policy for choosing the map partitioning requires further research that we plan to address as future work.

## 4. EXPERIMENTS

In order to facilitate experimentation, we created ACODA as a distributed platform based on JADE framework [4]. ACODA can be configured to run distributed ACO algorithms on an arbitrary network of computers.

The focus of the experiments presented in this paper was to analyze execution time and scalability of the configuration of ACODA for TSP with map partitioning. For that purpose we utilized the ACODA platform to run our version of the ACS algorithm, by varying the number of available machines from 1 to 7.

### 4.1 Setup

An experiment is structured as a fixed number of independent experimental rounds. A round consists of one execution of ACODA for a given initial set of parameters. A round is finished immediately after a fixed number of ants (given as parameter) were received by a given node of the map (known as reference node, also given as parameter). All parameters are initialized at the beginning of each round. Experimental data are collected during each round. At the end of the experiment (i.e. when the fixed number of rounds was reached) these data are post-processed to calculate experimental performance measures.

Setting-up and running ACODA on several computers assumes two stages: (i) initialization (or bootstrap) stage; and (ii) execution stage.

During the initialization stage: JADE platform is started, a number of containers are created (one container for each available machine), the nodes are prepared to be evenly distributed between agents, and finally the agents are created and evenly distributed on available containers of the JADE platform.

The experiment is ran during the execution stage. Execution is controlled using special control messages. Control messages are distinguished from ant exchange messages using their conversation id. Command messages have their conversation id set to “command”, while ant exchange messages have their conversation id set to “ant”. Command messages are given higher priority than ant exchanging messages.

An experimental round starts when all agents receive a “start” command, and ends when all agents receive a “stop” command. A designated agent called *Master* that manages the reference node, is responsible with controlling the experimental rounds (i.e. issuing of “start” and “stop” commands) and calculating performance measures. The *Master* counts the total number of ants that were received by the reference node. When this number reaches a given maximum value  $M$ , *Master* stops the current experimental round (issuing a “stop” command) and starts a new fresh experimental round (issuing a “start” command).

Duration  $T_i$  of an experimental round  $i$  is recorded by *Master* agent as time elapsed between issuing a “start” and

“stop” command. While the distributed ACO algorithm is running, minimum tour costs are passed from agent to agent via ants. Whenever an agent updates its current value of the minimum tour, the time elapsed since the agent received the “start” command is also recorded. When an experimental round is finished, *Master* agent saves the experimental data collected during the round.

Note that we cannot assume that the best tour recorded by the *Master* is actually the best tour computed during an experimental round. So we need to determine the minimum of the best tours computed by all agents together with the time at which this tour was discovered.

We are utilizing performance measures introduced in [9]. Let us suppose that we have  $n$  agents and  $r$  experimental rounds. For each agent  $j \in \{1, 2, \dots, n\}$ , let  $t_{i,j}$  be the time of the last update of the best tour performed by agent  $j$  in round  $i \in \{1, 2, \dots, r\}$ , and let  $v_{i,j}$  be the cost of the corresponding tour. *Master* agent will collect values  $v_{i,j}$  and  $t_{i,j}$  and will determine the solution  $v_i$  and associated time  $t_i$  in round  $i$  as shown in first two rows of table 2.

The experimental data that were acquired during all the rounds of an experiment are post-processed to calculate performance measures as shown in last five rows of table 2.

### 4.2 Experimental results

We ran several experiments using the following benchmark TSP maps selected from TSPLIB [11]: eil51, st70, kroA100, ch150, gr666. The number in the map name indicates the number of nodes of the map. These maps were chosen to experiment with different values of  $w_{avg}$  and  $\Delta w = w_{max} - w_{min}$  where  $\Delta w$  is the difference between the maximum and the minimum edge weight and  $w_{avg}$  is the average edge weight.

Similarly to [9], ACO parameters were set to the values recommended by [6] for ACS. The total number of ants is equal to the number of nodes  $n$  (i.e. for each node the ant population consists of a single ant),  $\tau_0 = 1/(n^2 w_{avg})$ ,  $\rho = \xi = 0.1$ ,  $\alpha = 1$ ,  $\beta = 5$ .

We define an ant move as the action of transferring an ant from a source node to destination node along a given edge. The number of ant moves was chosen according to the proposal from [6]. There, 1000 moves per ant were chosen for a 19 node map. Taking into account the sizes of our maps and scaling up proportionally, we set a number  $M = 10000$  of ant moves for our experiments.

We ran 10 rounds for each experiment on computer networks of 1, 4, and 7 computers with dual core processors at 2.5 GHz and 1GB of RAM memory. These workstations were interconnected using a high-speed Myrinet interconnection network at 2Gb/s. For each computer network configuration we ran three tests with variable number of nodes  $k \in \{1, 5, 10\}$  that are managed by each agent in the system.

A TSP map [11] identifies the nodes by numbers from 0 to  $n - 1$ . For the purpose of our experiments, we create  $q = \lfloor n/k \rfloor$  agents. Let  $p$  be the remainder of the division of  $n$  by  $k$ . When distributing nodes to agents, we use the order of node identifiers and assign each agent to the next  $k+b$  nodes, starting with node 0. In this distribution scheme  $b = 1$  for the first  $p$  agents and  $b = 0$  for the rest  $q-p$  agents. Therefore each agent manages either  $k$  or  $k+1$  nodes. Nodes of each agent are ordered according to their identifiers. For example if  $n = 51$  and  $k = 25$ , then  $p = 2$  agents must be created. The first agent will manage 26 nodes numbered 0 to

**Table 2: Calculation of performance measures.**

|                                                  |                                                                                                                                        |
|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| $v_i = \min_{j=1}^n v_{i,j}$                     | $v_i$ is the solution found by round $i$ .                                                                                             |
| $t_i = \min_{j=1}^n \{t_{i,j}   v_{i,j} = v_i\}$ | $t_i$ is the time in which solution was found by round $i$ .                                                                           |
| $v_{avg} = (\sum_{i=1}^r v_i)/r$                 | $v_{avg}$ is the average value of solutions found in all rounds.                                                                       |
| $t_{avg} = (\sum_{i=1}^r t_i)/r$                 | $t_{avg}$ is the average time in which solutions were found in all rounds.                                                             |
| $v_{min} = \min_{i=1}^r v_i$                     | $v_{min}$ is the best solution found after carrying out all rounds.                                                                    |
| $t_{min} = \min_{i=1}^r \{t_i   v_i = v_{min}\}$ | $t_{min}$ is the minimum time in which the best solution was found in all rounds; it indicates when the best solution was first found. |
| $T_{avg} = (\sum_{i=1}^r T_i)/r$                 | $T_{avg}$ is the average execution time of the rounds in an experiment.                                                                |

25, while the second agent will manage 25 nodes numbered 26 to 50.

The experiments (table 3) show a significant improvement in execution time when agents are managing more nodes. The number of nodes managed by an agent is determined as shown above, depending on the  $k$  chosen for the purpose of the experiment. As shown in table 3 it is clear that the execution time  $T_{avg}$  diminishes when  $k$  increases for every map regardless of the number of the computers in use. For example in the case of the map *eil51*, when running on a single computer, we obtained  $T_{avg} = 79$  for  $k = 1$  and  $T_{avg} = 53$  for  $k = 10$ .

Note that by including experiments with  $k = 1$ , we are comparing the results of the map partitioning approach with the version without map partitioning [9]. Moreover, the experiments (see table 3) clearly show that increasing the number of computers determines the decrease of the average execution time of the algorithm, so scalability of ACODA is preserved when each ACODA agent is managing more nodes. When applied to each map in the experiments, our approach showed a decrease in  $T_{avg}$  when increasing the number of computers. For example in the case of *kroA100* when  $k = 1$   $T_{avg}$  decreases from 274s when a single computer is used to 92s when 7 computers are used and the same happens for  $k = 10$ , a decrease from 136s, for 1 computer, to 80s, for 7 computers.

Speedup values in table 3 are calculated as  $T_{avg}$  for one node per agent on one computer over  $T_{avg}$  of the current line. Column "speedup" in table 3 clearly shows that when increasing the number of computers and the number of nodes managed by each agent we obtained better speedup values (see figure 1). The experiments also show that this method of improving efficiency is even more effective on bigger maps. For example, for map *eil51* we could only obtain a speedup of 2 at best, while on larger maps such as *ch150* and *kroA100* we could obtain a speedup of over 3.

Finally note that however, map partitioning has no immediate observable effect on the quality of the solutions. An explanation could be the fact that the total number of ant moves was the same for all experiments, with and without map partitioning. Nevertheless, more experiments are needed to clarify this issue.

## 5. RELATED WORK

TSP is a classic benchmark problem for heuristic search algorithms. With the advent of distributed computing technologies, distributed versions of heuristic algorithms for TSP

were also proposed. Based on our literature review, there are very few works that propose ACO-based distributed TSP algorithms. Moreover, there are even fewer proposals that utilize recent advances of multi-agent systems middleware for ACO-based TSP ([12]). Nevertheless, we could find references to multi-agent approaches to ACO for other combinatorial optimization problems ([15], [3], [5]).

A closely related approach to agent-based distributed ACO is presented in [12]. There, both graph nodes and ants are implemented as JADE agents. Ants centralize information about pheromone deposits and nodes' best tour cost through a single ACL message exchange per node [4]. This procedure adds up to  $2n$  messages per tour per ant, where  $n$  is the number of nodes. Each ant has to notify the node about its next hop and the cost of its tour in order for the node to be able to update its pheromone levels. This generates other  $n$  messages. When an ant completes a tour, it compares the tour cost with the collected best tours from the nodes. A best tour synchronization is triggered for all the nodes if a better tour has been found. This brings an additional overhead of  $n$  messages. So, [12] approach requires at most  $4n$  messages per tour per ant, while our approach requires at most  $2n$  messages:  $n$  messages (ant moves) to complete a tour and  $n$  messages to deposit the pheromone. We avoid the additional overhead of sending to nodes all the information necessary to carry out their tasks, as in ACODA this information is already contained in ant messages exchanged between nodes. Note that in this implementation of ACODA some of the messages are reduced to a simple enqueue-dequeue operations which is much faster than the jade messages between agents.

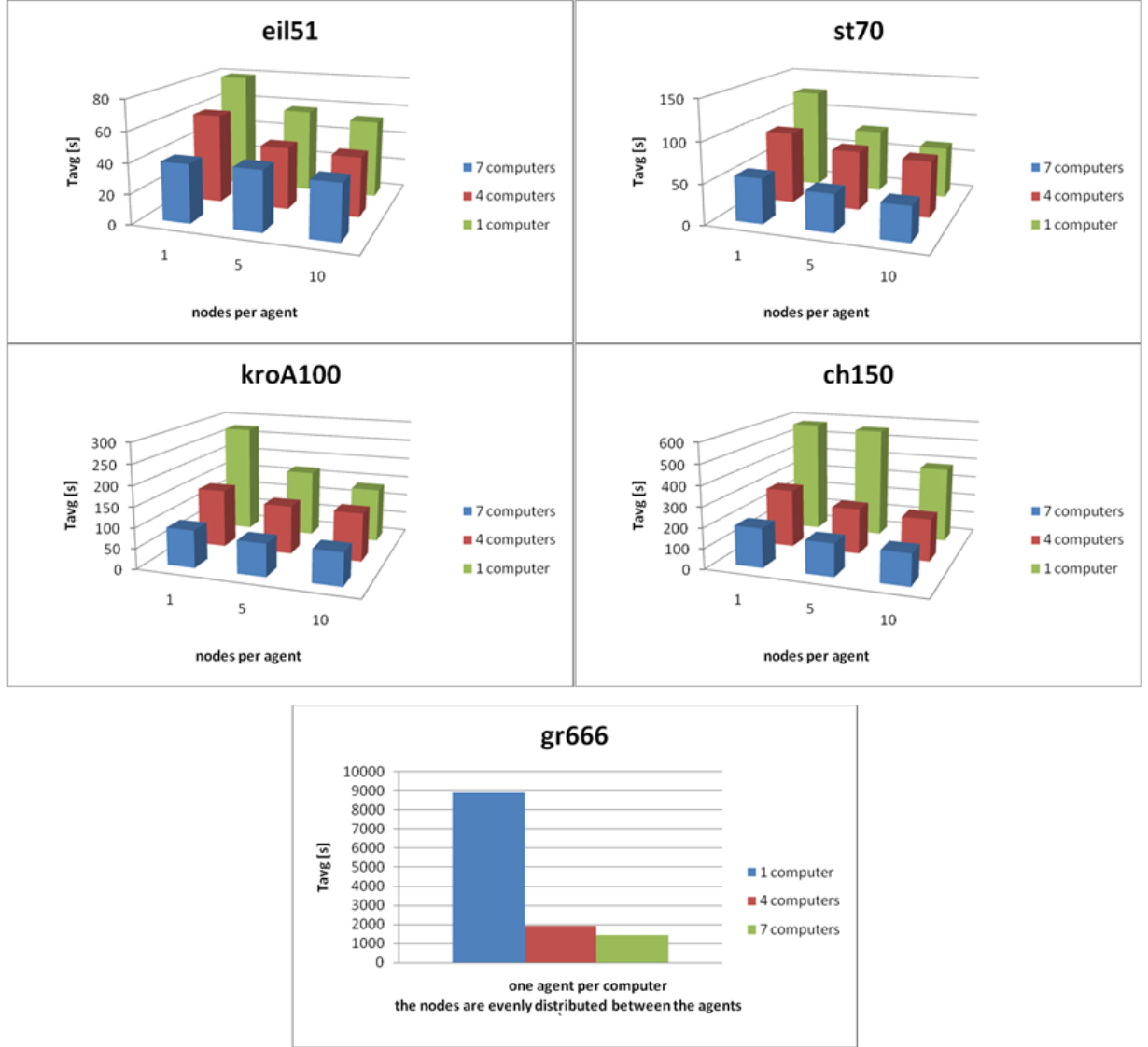
ACODA configuration reported here is based on the sequential ACS algorithm presented in [6]. There are however three notable differences: i) We avoid iterations. An iteration lasts until every ant has found a tour. It would be time consuming to synchronize all agents, so it would cancel the main benefits of an asynchronous, distributed architecture; ii) In ACS, best tours are compared after each iteration, thus allowing only the best ant to mark its tour. Again, this would require synchronization and centralization so we avoid it by allowing all ants to mark their tours; iii) In ACS ants move synchronously, taking one step at a time, while in our approach ants move asynchronously.

Papers [3] and [5] propose JABAT – a JADE-based middleware for agent teams. JABAT supports distributed implementation and collaboration of population-based optimization algorithms. In particular, JABAT was applied to

**Table 3: Experimental results.** The number in the name of the map represents the number of nodes and the number written between brackets represents the optimum tour cost. Speedup values are calculated as  $T_{avg}$  for one node per agent on one computer over  $T_{avg}$  of the current line.

| map            | computers | k   | $v_{avg}$ | $t_{avg}[s]$ | $v_{min}$ | $t_{min}[s]$ | $T_{avg}[s]$ | speedup |
|----------------|-----------|-----|-----------|--------------|-----------|--------------|--------------|---------|
| eil51(426)     | 1         | 1   | 459.2     | 33           | 452       | 27           | 79           | 1       |
|                | 1         | 5   | 489.6     | 18           | 478       | 20           | 57           | 1.3     |
|                | 1         | 10  | 480       | 16           | 475       | 18           | 53           | 1.4     |
|                | 4         | 1   | 462.4     | 46           | 449       | 64           | 60           | 1.3     |
|                | 4         | 5   | 485.8     | 18           | 479       | 16           | 42           | 1.8     |
|                | 4         | 10  | 469.8     | 23           | 461       | 3.9          | 40           | 1.9     |
|                | 7         | 1   | 449.4     | 13           | 442       | 10           | 39           | 2       |
|                | 7         | 5   | 480.8     | 13           | 472       | 25           | 40           | 1.9     |
|                | 5         | 10  | 471.6     | 15           | 467       | 17           | 37           | 2       |
| st70(675)      | 1         | 1   | 729.4     | 67           | 712       | 38           | 127          | 1       |
|                | 1         | 5   | 786.6     | 16           | 772       | 41           | 80           | 1.5     |
|                | 1         | 10  | 779       | 20           | 766       | 47           | 65           | 1.9     |
|                | 4         | 1   | 720.4     | 32           | 701       | 45           | 90           | 1.4     |
|                | 4         | 5   | 786.2     | 20           | 784       | 26           | 74           | 1.7     |
|                | 4         | 10  | 777       | 24           | 757       | 55           | 70           | 1.8     |
|                | 7         | 1   | 688.6     | 25           | 684       | 6.4          | 56           | 2.2     |
|                | 7         | 5   | 752.3     | 20           | 702       | 17           | 47           | 2.7     |
|                | 7         | 10  | 771.4     | 32           | 691       | 40           | 43           | 2.9     |
| kroA100(21282) | 1         | 1   | 22590.8   | 38           | 22152     | 31           | 274          | 1       |
|                | 1         | 5   | 22443.8   | 37           | 21847     | 136          | 168          | 1.6     |
|                | 1         | 10  | 22554.6   | 86.9         | 21735     | 110          | 136          | 2       |
|                | 4         | 1   | 22677.8   | 22           | 22232     | 15           | 145          | 1.9     |
|                | 4         | 5   | 22513.2   | 32           | 21896     | 103          | 121          | 2.2     |
|                | 4         | 10  | 22625.2   | 84           | 21930     | 115          | 120          | 2.3     |
|                | 7         | 1   | 22997     | 29           | 22605     | 77           | 92           | 2.9     |
|                | 7         | 5   | 22141     | 20           | 21748     | 16           | 80           | 3.4     |
|                | 7         | 10  | 22319.8   | 45           | 21937     | 39           | 80           | 3.4     |
| ch150(6528)    | 1         | 1   | 7969.3    | 257          | 6992      | 524          | 572          | 1       |
|                | 1         | 5   | 7572.6    | 139          | 6974      | 149          | 558          | 1       |
|                | 1         | 10  | 7166      | 145          | 6922      | 167          | 380          | 1.5     |
|                | 4         | 1   | 7650.8    | 135          | 7434      | 281          | 291          | 1.9     |
|                | 4         | 5   | 7771      | 62           | 7013      | 98           | 227          | 2.5     |
|                | 4         | 10  | 7503.7    | 57           | 7048      | 82           | 211          | 2.7     |
|                | 7         | 1   | 7729.4    | 17           | 7638      | 26           | 194          | 2.9     |
|                | 7         | 5   | 7601.8    | 37           | 6851      | 43           | 161          | 3.5     |
|                | 7         | 10  | 7594.2    | 32           | 6965      | 52           | 155          | 3.7     |
| gr666(294358)  | 1         | 666 | 311931    | 3229         | 307038    | 1122         | 8917         | 1       |
|                | 4         | 166 | 307440    | 688          | 301808    | 596          | 1920         | 4.6     |
|                | 7         | 95  | 312352    | 398          | 309416    | 1268         | 1458         | 6.1     |

Figure 1: Graphs of the execution time  $T_{avg}$  (the time elapsed until each node has moved 10000 ants).



TSP. There is however a very important difference between JABAT and ACODA. JABAT agents represent sequential improvement algorithms that cooperate for solving a problem in a distributed way. ACODA agents provide a natural distributed model of the problem environment that is suitable for distributed ACO algorithms. Moreover, we could not find scalability studies referring to JABAT.

Paper [15] proposes a JADE-based multi-agent environment for dynamic manufacturing scheduling that combines intelligent techniques of ACO and multi-agent coordination. However, the focus in [15] is to evaluate the impact of ACO intelligence on multi-agent coordination, rather than to utilize multi-agent middleware for improving ACO. Therefore ACO intelligence is embedded into job and machine agents that have the role of ants, which is different from ACODA where ants are passive objects exchanged by agents providing a distributed model of the problem environment.

In [7] authors compare a distributed form of ACS with flooding algorithm applied on resource discovery problem.

They experiment with both algorithms using *ns-2* network simulation tool ([1]). The down side of this is that no real execution time measure can be made using their approach. They showed that ACS is the better approach in terms of: best success rate, least number of hops and least traffic. The detailed algorithm and ACO parameters are not presented in order to duplicate their approach using our ACODA framework, for a realistic comparison. The differences between their approach and ACODA are: (i) resource queries are handled centrally at a single anthill, thus introducing single point of failure, (ii) they do not take edge weights into account as they are trying to solve the resource discovery problem, not the TSP problem, and (iii) ants are implemented as *ns-2* mobile agents, while in ACODA we are using JADE. Moreover, in practice there is no need for code mobility as every ant is governed by the same behavioral rules. So, in our approach we use “nodes as agents” to control ants’ movement using messages, rather than using code (i.e. JADE agent) mobility.

Paper [8] proposes purely theoretical frameworks for multi-agent systems. No experiments or implementations are mentioned. The authors present a distributed form of ACO based on so called “smart messages” approach to multi-agent systems where agent mobility is used to implement complex communication over dynamic networks. They use delegate multi-agent systems to manage these smart messages in order to design a multi-agent approach for ACO. They do not present ways of representing the environment, determining convergence or stopping condition of ACO experiments.

In [13] authors thoroughly analyze multi-colony ACO algorithms applied on TSP. The main difference from our approach is that [13] proposes to run a separate colony on each available processor. At predetermined time points, solutions are centrally collected and local search is performed on the best solution provided by each colony. This distributed ACO approach is not fully decentralized, as it requires synchronization and centralized sequential local search after each solution was centrally collected. Our approach is fully decentralized, parallel and asynchronous, and thus it could lend itself to heavy parallelization.

In [2] the authors present a neural network approach for optimal partitioning of TSP maps. The idea was to improve efficiency by choosing an optimal partitioning of the TSP map in such a way that by combining tours for each sub-map of the partition will result in a nearly optimal cost tour. The differences from our approach are: i) our approach is distributed, while no attempt for distribution is made in [2]; ii) whatever initial arbitrary partitioning scheme we use, our algorithm can still find good results as our ants are not restricted to find optimal tours for each sub-map of the partition; rather, they can freely move from one sub-map (agent) to another when needed; iii) we use ACO on the whole map, rather than applying the heuristic on each individual sub-map of the partition.

## 6. CONCLUSIONS

In this paper we presented experimental results of applying our multi-agent ACODA framework for distributed ACO for solving TSP using a map partitioning method. In the experiments we utilized a distributed version of the ACO approach, starting from the initial proposal of the ACS system. Experimental results on sample benchmark TSP problems were encouraging, as they clearly support the scalability of our system, as well as the improvement of the execution time compared to our initial work [9] (that did not employ map partitioning). Some directions that we would like to follow as future work: i) strengthen the results by experimenting with larger TSP problems on larger and faster computer networks, as well as with different policies for the map partitioning (currently a simple straightforward map partitioning method was investigated); ii) support the generality of our framework by considering other forms of distributed ACO.

## 7. ACKNOWLEDGMENTS

This work was partially supported by the strategic grant POSDRU/88/1.5/S/50783, Project ID50783 (2009), co-financed by the European Social Fund – Investing in People, within the Sectoral Operational Programme Human Resources Development 2007 - 2013.

## 8. REFERENCES

- [1] The ns-2 project, a network simulation tool. <http://nsnam.isi.edu/nsnam/index.php/>.
- [2] N. Aras, I. K. Altinel, and J. Oommen. A kohonen-like decomposition method for the traveling salesman problem. In *ECAI 2000. 14th European Conference on Artificial Intelligence*. IOS Press, 2000.
- [3] D. Barbucha, I. Czarnowski, P. Jedrzejowicz, E. Ratajczak, and I. Wierzbowska. Jade-based a-team as a tool for implementing population-based algorithms. In *Proc. 6th International Conference on Intelligent Systems Design and Applications: ISDA '2006*, pages 144–149. IEEE Computer Society, 2006.
- [4] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons Ltd, 2007.
- [5] I. Czarnowski, P. Jedrzejowicz, and I. Wierzbowska. A-team middleware on a cluster. In *Proc. 3rd KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications: KES-AMSTA '2009*, volume 5559, pages 764–772. Springer-Verlag, 2009.
- [6] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.
- [7] S. M. Fattahi and N. M. Charkari. Distributed acs algorithm for resource discovery in grid. *Special Issue of the International Journal of the Computer, the Internet and Management*, 17(SP1), 2009.
- [8] T. Holvoet, D. Weyns, and P. Valckenaers. Patterns of delegate mas. volume 0, pages 1–9, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [9] S. Ilie and C. Bădică. Distributed multi-agent system for solving traveling salesman problem using ant colony optimization. In *Proc. 4th International Symposium of Intelligent Distributed Computing, IDC'2010*, volume 315, pages 119–129. Springer-Verlag, 2010.
- [10] S. Ilie and C. Bădică. Effectiveness of solving traveling salesman problem using ant colony optimization on distributed multi-agent middleware. In *Proc. of the International Multiconference on Computer Science and Information Technology, IMCSIT 2010*, pages 197–203, 2010.
- [11] G. Reinelt. TspLib - a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.
- [12] E. Ridge, D. Kudenko, and D. Kazakov. Parallel, asynchronous and decentralised ant colony system. In *In Proc. of the First International Symposium on Nature-Inspired Systems for Parallel, Asynchronous and Decentralised Environments (NISPADE)*, 2006.
- [13] C. Twomey, T. Stützle, M. Dorigo, M. Manfrin, and M. Birattari. An analysis of communication policies for homogeneous multi-colony aco algorithms. *Information Sciences*, 180(12):2390–2404, 2010.
- [14] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons Ltd, 2002.
- [15] W. Xiang and H. P. Lee. Ant colony intelligence in multi-agent dynamic manufacturing scheduling. *Engineering Applications of Artificial Intelligence*, 21(1):73–85, 2008.