Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 611 – 620

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Multi-agent distributed framework for swarm intelligence

Sorin Ilie[a], Costin Bădică[a]

[a]*University of Craiova, Software Engineering Department, Bvd. Decebal 107, Craiova 200440, Romania*

Abstract

This paper presents a multi-agent distributed framework for Swarm Intelligence (SI) based on our previous work ACODA (Ant Colony Optimization on a Distributed Architecture). Our framework can be used to distribute SI algorithms for solving graph search problems on a computer network. Examples and experimental results are given for SI algorithms of: Ant Colony System (ACS) and Bee Colony Optimization (BCO). In order to use the framework, the SI algorithms must be conceptualized to take advantage of the inherent parallelism determined by their analogy with natural phenomena (biological, chemical, physical, etc.): (i) the physical environment of the swarm entities is represented as a distributed multi-agent system and (ii) entities' movement in the physical environment is represented as messages exchanged asynchronously between the agents of the problem environment. We present initial experimental results that show that our framework is scalable. We then compare the results of the distributed implementations of BCO and ACS algorithms using our framework. The conclusion was that our approach scales better when implementing the ACS algorithm but is faster when implementing BCO.

Keywords: Distributed Applications; Swarm Intelligence; Artificial Intelligence; Ant Colony Optimization; Bee Colony Optimization

1. Introduction

This paper introduces a new distributed framework for a class of Swarm Intelligence algorithms that better exploits the inherently distributed nature of these algorithms.

Swarm Intelligence (SI hereafter) "is the emergence of coherent functional global patterns from the collective behaviors of entities interacting locally with their environment" [1].

SI approaches are usually applied to solve computationally complex problems, such as NP-hard problems. There is a certain level of abstraction at which the solving of such problems can be modeled by distributed computational systems composed of interacting artificial entities. Thus we expect distributed computing, including distributed multi-agent middleware, to have great potential for the application of SI computational approaches.

The framework presented here is an extension of our ACODA (Ant Colony Optimization on a Distributed Architecture) which was initially designed to support only Ant Colony Optimization (ACO). Initial experimental results on the scalability of ACODA were already presented in [2].

In fact our framework can be applied to a class of SI algorithms with the condition of abstracting the process of searching the solution space as the movement of computational swarm entities into their distributed computational

*Corresponding author. Tel. and fax: +40-251-438.198.
E-mail address: cbadica@software.ucv.ro.

doi:10.1016/j.procs.2013.05.225

environment, similarly with the movement of natural entities (like birds, ants, or bees) in their physical environment. During their movement process, these entities save partial solutions as well as other relevant information onto the environment, sense the environment and exchange information via the environment, thus achieving the continuous improvement of partial solutions into gradually better solutions. Typical examples of SI approaches suitable for our framework are ACO and Bee Colony (BC). In particular, we considered here the algorithms of Ant Colony System (ACS) presented [3] in chapter 3 and Bee Colony Optimization (BCO) presented in [4].

In this paper we present our framework and show how it can be used to implement distributed ACO and BC. Moreover, we updated the initial design of the framework presented in our previous work [2] in order to accommodate the implementation of more types of SI algorithms and problems.

Summarizing, the main contributions of this paper can be outlined as follows:

(i) We generalized our ACODA distributed framework, initially designed only for ACO algorithms, to other types of SI algorithms. The novelty is the new modeling of the SI environment as a multi-agent system and conceptualizing the mobility of swarm entity as messages exchanged by agents.
(ii) We showed that the new framework supports two different types of SI: ACS [3] and BCO [4].
(iii) We performed an experimental comparison of the implementations of BCO and ACS for solving the Traveling Salesman Problem (TSP hereafter) on our framework. In our experiments with BCO we obtained the best value of efficiency (equation (1)) that we found in literature.

The paper is structured as follows. In section 2 we introduce *graph search problems* that are targeted by our framework and we give an overview of ACS and BCO. In section 3 we present our literature review on distributed SI. Section 4 presents our framework. While in section 5 we show how ACS and BCO can be implemented on it. In section VI we present our experiments and in section 7 we present our conclusions and future work.

2. Background

Before presenting our architecture design, we state the target problem and introduce the SI approaches that we used as case studies. Due to lack of space we will not detail the mathematical engines of the implemented SI algorithms, as they are known and can be found in in [3] (ACS) and in [4] (BCO).

SI algorithms are well suited for graph search problems, so our framework targets this type of problems. A *graph search problem* can be formalized as optimizing a real-valued function defined over a subset of graph paths – i.e. paths restricted by specific problem-dependent constraints. For the TSP this subset contains all the Hamiltonian cycles [5] of the graph – i.e. paths containing all vertices such that the source and the destination are the same, while any other two vertices of the path are distinct. Other notable examples of graph search problems include: graph flow problems [5], generalized TSP [6], pickup and delivery problem [7], etc.

2.1. Ant Colony Optimization

ACO [3] refers to a family of SI optimization algorithms that get their inspiration from the metaphor of real ants searching for food. During their searching process, ants secrete pheromone to mark their way back to the anthill. Other colony members sense the pheromone and become attracted by 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.

In the ACO approach to SI, artificial ants are programmed to mimic the behavior of real ants while searching for food. The ants' environment is modeled as a graph while the path to the food becomes the solution to a given graph search problem. Artificial ants originate from the anthills that are vertices of the graph and travel between vertices to find optimal paths, following ACO rules. When a solution path is found, ants mark its edges with pheromone by retracing the path.

We mapped the ACS algorithm [3] to our framework by configuring it with the parameters recommended by the authors. In ACS an ant chooses the next edge to follow towards an unvisited vertex with a probability that increases with the quantity of pheromone deposited on that edge.

2.2. Bee Colony

BC refers to a family of SI optimization algorithms that get their inspiration from the metaphor of real bees searching for food. Bee colonies forage for pollen sources by moving randomly in the physical environment. When a bee finds a rich food source, upon its return to the hive, it starts to perform the so called "waggle dance" [8]. The purpose of this dance is to inform the other bees about the direction and distance to the food source [9]. The other bees from the swarm will be attracted to explore the indicated location.

In BC artificial bees are programmed to mimic the behavior of real bees while searching for food. There are many types of BC computational approaches proposed in the literature [10], [11]. In this paper we adapted the specific version of the BCO algorithm for solving TSP introduced in [4] to our distributed SI framework. The bees environment is modeled as the TSP graph while the path to the food becomes the solution to a given graph search problem. Intuitively, a bee determines the next vertex where to go by analyzing the currently best solution known by the colony and determining a probability for moving to an unvisited neighboring vertex. Assuming that the bee is currently placed in vertex i and that the successor of i in the currently best solution is j, the bee computes a probability $\lambda > 0$ for moving to j. If there is no currently best solution known yet by the colony or if j was already visited by the bee then $\lambda = 0$. For all the other unvisited neighbor vertices k of i the bee computes the probability of moving to k by equally distributing the remaining probability $1 - \lambda$ to each of the unvisited neighboring nodes.

A local search algorithm [12] can be used to improve the quality of the solutions offered by both ACS and BCO. An example of local search algorithm is *2-opt*. It takes a solution, chooses two distinct edges of the path representing the solution and tries to reorganize them in such a way that the quality of the solution improves.

2.3. Speedup and Efficiency

Performing a fair comparison of the different approaches for distributing various SI algorithms requires the use of general and meaningful quantitative measures that are independent on the distribution model. For this reason we introduce the classical notions of *speedup* and *computational efficiency*.

The *speedup* metric S_c of a distributed application measures how much the application is faster when running on more computing nodes as compared to running on a single computing node. We define the term *computing node* as abstraction of a computer or machine, physical or virtual, with its own or allocated processor and memory. This term is used in the literature on distributed applications [13] and adopted by distributed computing support systems, including for example the Condor cluster workload management system [14] that we used in our experiments. Very often the short version "node" is used in the literature, but this can be confused with "vertex" from graph theory [5]. Therefore we preferred to use the longer version "computing node" to avoid confusions.

The *computational efficiency* metric e_c normalizes the value of the speedup to the number of computing nodes. The two metrics are introduced by equation (1).

$$S_c = \frac{T_1}{T_c} \qquad e_c = \frac{S_c}{c} \tag{1}$$

In equation (1): c is the number of computing nodes; S_c is the speedup of the application; T_1 is the execution time of the application on a single computing node; T_c is the execution time of the application on c computing nodes; e_c is the efficiency of running an application on c computing nodes.

3. Related Work

A very recent overview and classification of parallel computing approaches to Ant Colony Optimization (ACO) was reported by [15]. The authors propose an interesting and novel classification scheme for parallel ACO algorithms. The classification includes: master-slave model, cellular model, independent runs model, island model and hybrid models. However, the authors do not include in their analysis the agent-based approaches by arguing that they are usually not designed to take advantage of multi-processor architectures in order to reduce execution time. A detailed state of the art of distributed approaches for ACO has already been presented in [2].

The cellular model mentioned in [15] is actually based on the author's own work [16] which proposes the splitting of ACO search space into overlapping neighborhoods, each one with its own pheromone matrix. The good solutions gradually spread from one neighborhood to another through diffusion. This approach requires the partitioning of the search space such that each set contains a continuous part of the optimal solution. However, this condition is almost never met in practice, so an optimal solution is hardly reachable. The authors are the only ones that have ever implemented and tested this model.

In the case of BC approach to SI we could find papers [17] and [18] that collectively offer a good classification of BC parallel approaches. Interestingly, these papers identify the same general taxonomy of approaches for BC as in [15] does for ACO, i.e. master-slave model, independent runs model, island model and hybrid models.

Our review of the literature shows that most authors focused on the island model. The *island model* of distributing SI requires running a separate group of entities, referred to as an "island", on each available computing node. Each island has its own copy of the search space, i.e. a graph, so it can be considered in fact a sequential implementation of the SI algorithm. At predetermined points in time, with fixed frequency (i.e. periodically), islands communicate solutions to each other and the best solution over all is marked as required by the SI algorithm. Depending on the implementation, solutions can also be communicated asynchronously.

The results reported in the literature generally obtained top efficiency when distributing their island model implementations on 8 or less computing nodes. The top efficiency we are aware of was reported in [19] for asynchronous ACO approaches using the island model that boast 0.9 efficiency on 8 nodes. The results obtained on the TSP used graphs from the benchmark library TSPLIB [20] of up to 500 vertices. We did, however, find one approach that reports scalability beyond 8 computing nodes in [21]. In that paper the authors presented an ACO island model that reached peak efficiency at 25 nodes. Note however that the efficiency obtained was inferior to that reported in [19]. This result was tested on a 318 vertex TSP instance.

We found significant research sustaining the superiority of asynchronous communication for the island model. For example, paper [22] compared three approaches to BC: an island model using sequential communication, one using asynchronous communication and a sequential version of BC. The conclusions drawn by the authors were that the asynchronous distributed approach is superior to the others in both quality of solutions and execution time.

Our conclusion supported by a number of papers from the literature on this subject, as well as by their positive results is that the island approach represents the state of the art model for distributing SI computations. However, the island model requires the sequential implementation of the SI algorithm in each island. So the authors of island model implementations do not take advantage of the inherently distributed nature of SI approaches.

Agent-based approaches to SI tend to be purely theoretical such as [10] for BC and [23] for ACO. We could also find some papers that actually report implementations of SI using agents. Note however that they usually do not provide a study of scalability and efficiency [15]. One example is the work reported in paper [24]. It presents a BCO-inspired agent based approach called "BeeJamA" to the Vehicle Routing Problem. However, the approach was evaluated only in terms of solution quality using a multi-agent simulation toolkit.

4. Development of a Distributed Framework for SI Algorithms

Our framework takes a representation of entities' environment as input data, i.e. the graph in this case, and creates a distributed computing environment onto the available computing nodes, based on this representation. The distributed environment is then populated with software agents. Each agent is assigned a subset of the graph vertices representing a part of the graph, as described in paper [2].

Each agent creates its own population of entities represented as software objects that can be transferred from vertex to vertex with the help of agents, thus mimicking migration of natural entities in their natural environment. When an entity must migrate between two vertices managed by different agents, the object is serialized and sent via a message. Otherwise, the migration is achieved by pushing and popping the object from a local agent queue.

We have implemented our distributed platform using JADE multi-agent framework [25]. The UML class diagram that shows the most important classes of our framework is presented in Fig. 1. The central class of the framework is MOBILE-AGENT. This is an abstract class that represents an agent of the framework that manages a subset of the graph vertices. A MOBILE-AGENT is able to control the movement of entities between two graph vertices, either locally or by exchanging messages with other agents of the framework. Additionally, a MOBILE-AGENT is able to interpret special messages that control the execution of the framework.

The execution of an agent in our framework follows the JADE model [25] i.e. it assumes the non-preemptive interleaved execution of a set of behaviors. A behavior is conceptually a single execution unit of an agent and it is represented by class BEHAVIOR (see Fig. 1). MOBILE-AGENT class contains three behaviors:

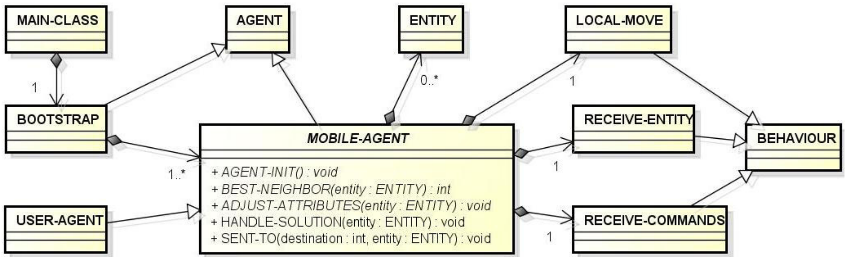

MAIN-CLASS
AGENT
ENTITY
LOCAL-MOVE
BOOTSTRAP
MOBILE-AGENT
RECEIVE-ENTITY
BEHAVIOUR
USER-AGENT
RECEIVE-COMMANDS
+ AGENT-INIT() : void
+ BEST-NEIGHBOR(entity : ENTITY) : int
+ ADJUST-ATTRIBUTES(entity : ENTITY) : void
+ HANDLE-SOLUTION(entity : ENTITY) : void
+ SENT-TO(destination : int, entity : ENTITY) : void
1
0..*
1
1..*
1
1

Fig. 1. Framework class hierarchy.

(i) RECEIVE-ENTITY that handles incoming entities received as messages sent by other agents;
(ii) LOCAL-MOVE that locally handles the entities either by pushing them onto the local queue or by popping them out from the local queue;
(iii) RECEIVE-COMMANDS that handles incoming messages representing commands for controlling the execution of the framework.

The MOBILE-AGENT abstract class provides four abstract methods:

(i) AGENT-INIT(): this methods is called once when the agent starts. The main purpose of this method is to initialize the entity population of the agent.
(ii) BEST-NEIGHBOR(entity): this method determines the next vertex the entity should visit in order to get closer to completing a solution.
(iii) ADJUST-ATTRIBUTES(entity): this method implements the logic necessary for adjusting entity attributes.
(iv) HANDLE-SOLUTION(entity): this method is called every time an agent receives a complete solution from other agents. This is meant to be used in the case of SI algorithms that require the broadcasting of solutions such as BCO (i.e. the "waggle dance"). The implementation of this method is not always necessary because some SI algorithms such as ACS do not need it (see also Fig. 2).

These methods must be implemented by the user of the framework in a class derived from MOBILE-AGENT (shown as class USER-AGENT on Fig. 1) to support a specific SI algorithm.

The class MOBILE-AGENT is also using a method SEND-TO(destination, entity) in order to send an entity to a given destination vertex. Finally, MOBILE-AGENT is a basic agent class providing the mobility capability that is required during the framework setup process (see below).

Each entity is represented by the ENTITY class that contains the members: cost of the currently explored path; returning flag; vertex of origin; best solution cost that the entity knows, based on its search history; a list of vertices representing the path followed to reach the current location; a method that resets all members to their initial values, i.e. initializes the entity (this is useful for reusing an ENTITY object).

The setup of the framework requires the instantiation of a JADE platform, as well as the creation of an agent container running on each computing node. For this purpose we developed MAIN-CLASS with two functions:

(i) to create the Main-Container required by the JADE platform and then to create the BOOTSTRAP agent that will be used for setting up the framework.
(ii) to remotely create auxiliary JADE containers, one container on each available computing node.

The BOOTSTRAP agent that is created on the JADE Main-Container has the following roles:

(i) to create the framework agents, of class USER-AGENT;
(ii) to allocate the vertices of the problem graph to each agent;
(iii) to use JADE mobility to distribute these agents onto the available computing nodes, identified by their corresponding agent containers.

By default, one agent is allocated to each container and the set of vertices is equally distributed to the agents.

SI algorithms are heuristic by nature and as such, they require multiple runs of the same experiment in order to reach a sound statistical conclusion. That is why the MOBILE-AGENT running on the Main-Container (called

Table 1. Calculation of performance measures.

Parameter	Explanation
$v_i = \min_{j=0}^{n-1} v_{i,j}$	v_i is the solution found by round i.
$t_i = \min_{j=0}^{n-1}\{t_{i,j} \| v_{i,j} = v_i\}$	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.
T_i	T_i is the duration of experimental round i.
$T_{avg} = (\sum_{i=1}^{r} T_i)/r$	T_{avg} is the average execution time of the rounds in an experiment.

Master) has the additional role of controlling the experiment and gathering the experiment statistics. Experiments' execution is controlled using special control messages that are distinguished from the entity exchange messages (see [2] for more details). MOBILE-AGENT handles such messages using RECEIVE-COMMANDS behavior.

An experiment is organized as a fixed number r of rounds. Each of them consists of one execution of the SI algorithm for a given set of parameters and it ends when the stop condition is met. The SI algorithm's parameters are reset at the beginning of each round using AGENT-INIT(). Experimental data is collected during each round. At the end of the experiment this data is post-processed to calculate performance measures.

Using the framework for the definition and running of an SI algorithm requires the following steps:

(i) the setup of one JADE container running on each computing node
(ii) the preparation of the code of the SI algorithm and the definition of its parameters
(iii) the definition and coding of the stop condition. The default stop condition is true when a predefined number M of entities visited the "reference vertex".

Note that by allocating all vertices to a single agent (i.e. using a single computing node) no entity will be transferred to another agent. All entities will only use the queue of the LOCAL-MOVE behaviour resulting in a sequential implementation of the SI algorithm.

4.1. Performance Measures

The standard approach for analyzing the performance of heuristic approaches such as SI is to perform statistical analysis on a population of multiple sets of data collected during repeated executions of the same test [1]. Therefore we organized our experiments as a fixed number of independent rounds. Each of them consists of one execution of the SI algorithm for a given set of parameters and it ends as soon as a stop condition is met.

Experimental data for performance measures' evaluation were captured during each experiment. Duration T_i of an experimental round i is recorded by the Master agent. Whenever an agent updates its current value of the best solution, the time elapsed since the start of the experiment is recorded. When an experimental round is finished, Master agent gathers the raw statistical data from all the agents. This data is then used by the Master agent to calculate performance measures of the current round.

We utilized the performance measures introduced in [2] and we briefly review them here. Let us suppose that we have n vertices and r experimental rounds. For each vertex j let $t_{i,j}$ be the time of the last update of the best solution found by agent j in round i and let $v_{i,j}$ be the cost of this solution. The experimental data that is acquired during all the rounds of an experiment are post-processed to calculate performance measures as shown in Table 1.

Please note that our primary concern was to improve the distribution of the computation performed by existing SI algorithms by exploiting the inherent parallelism determined by their analogy with natural phenomena (biological, chemical, physical, etc.). So evaluation was done mainly in terms of execution time and scalability.

5. Using the Framework

In order to implement an SI algorithm for solving a specific graph search problem (like TSP), the following four methods must be implemented after extending the MOBILE-AGENT abstract class: AGENT-INIT(), HANDLE-SOLUTION(), BEST-NEIGHBOR() and ADJUST-ATTRIBUTES().

We will now show how these methods can be implemented for the ACS and BCO algorithms. Then experimental results will be presented for each case.

5.1. ACS Implementation

In this implementation, entities of the framework represent artificial ants, as found in ACO algorithms. Fig. 2 outlines the ACS implementation in our framework. Please note that only the methods that should be implemented by the user are shown.

```
AGENT-INIT()
1. $V \leftarrow$ MANAGED-VERTICES()
2. for $v \in V$ do
3.     CREATE-ENTITY($v$)

ADJUST-ATTRIBUTES(ant)
1. if AT-ANTHILL(entity) then
2.     if RETURNING(entity) then
3.         INITIALIZE(entity)
4.     if SOLUTION-COMPLETED(entity) then
5.         SET-RETURN-FLAG(entity)
6.         LOCAL-SEARCH(entity)
6.         CALC-PHEROMONE-STRENGTH(entity)
7. UPDATE-BEST-SOLUTION(entity)

BEST-NEIGHBOR(entity)
1. if RETURNING(entity) then
2.     if BEST-SOLUTION(entity) then
3.         DEPOSIT-PHEROMONE()
4.     else return ANTHILL(entity)
5.     return LAST-VISITED-NODE(entity)
6. bestNeighbor ← RANDOM-CHOICE()
7. LOCAL-EVAPORATE-PHEROMONE()
8. return bestNeighbor
```

Fig. 2. ACS implementation.

AGENT-INIT() method creates one ant per vertex. BEST-NEIGHBOR() determines the vertex where an ant should be sent. However, if an ant has found a solution and it is returning then the ant has to retrace its path and mark it with pheromone. In this case BEST-NEIGHBOR() sends the ant back to the last visited vertex, which is popped from the list of visited vertices, and deposits the ant's pheromone on the corresponding edge. In this example a solution is marked with pheromone only if it has at least the quality of the best known solution so far. BEST-NEIGHBOR() also implements pheromone evaporation. ANTHILL() returns the identifier of the agent that manages the anthill, i.e. the vertex of origin of the ant.

SOLUTION-COMPLETED() method returns true if the path followed by the ant is a solution. ADJUST-ATTRIBUTES() method sets the returning flag (if necessary), performs local search on the ant's path and calculates pheromone strength whenever an ant has completed a solution. Ants that have returned after completing a solution are refreshed. All agents maintain locally the value of the best known solution. Therefore, whenever an ant moves, the agent that is managing the vertex where the ant is located updates its local best known solution using the UPDATE-BEST-SOLUTION() method. Whenever an ant visits this vertex, it is able to sense the environment and eventually to update its "current knowledge" about the value of the best solution found so far. Hence agents have the possibility to send and receive via the ants not only pheromone information, but also qualitative information about the best solutions found so far.

The HANDLE-SOLUTION() method does not need to do anything since in ACS pheromone updates are done automatically when ants decide where to move, inside BEST-NEIGHBOR() method.

5.2. BCO Implementation

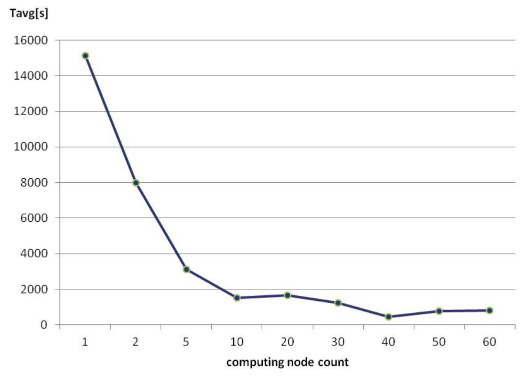
In this implementation, entities of the framework represent artificial bees, as found in BC algorithms. Fig. 3 outlines the BCO implementation in our framework.

ADJUST-ATTRIBUTES() method performs local search on the bees that have found a solution, broadcasts them to the other agents (i.e. the "waggle dance" is performed) and then these bees are refreshed. As in the case of ACS, only the entities that improve the best known solution are allowed to perform the "waggle dance". All agents maintain the best known solution locally. The HANDLE-SOLUTION() method is called whenever the agent receives the broadcasted bees. The agent then updates the best known solution using method UPDATE-BEST-SOLUTION().

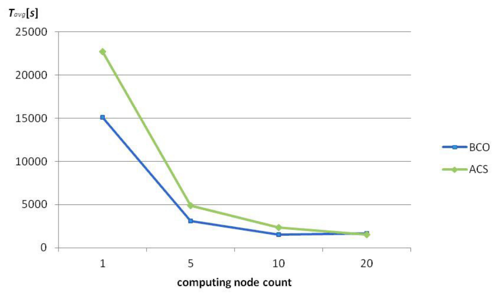
Comparing BCO with ACS implementations we can easily conclude that the main difference is the method used by the agents to notify each other about the solutions found. In ACS the entities mark the solutions with pheromone while retracing their path resulting in additional σ entity moves, where σ is the size, i.e. the number of vertices, of the solution found by that entity. When more than one agent is used, some of those moves will inevitably be messages exchanged by agents. On the other hand BCO sends a single broadcast message with the

```
AGENT-INIT()
1. $V \leftarrow$ MANAGED-VERTICES()
2. **for** $v \in V$ **do**
3.     CREATE-ENTITY($v$)

ADJUST-ATTRIBUTES(*entity*)
1. **if** SOLUTION-COMPLETED(*entity*) **then**
2.     LOCAL-SEARCH(*entity*)
3.     BROADCAST-SOLUTION(*entity*)
4.     INITIALIZE(*entity*)

BEST-NEIGHBOR(*entity*)
1. bestNeighbor¡= RANDOM-CHOICE()
2. ADD-TO-VISITED-LIST(bestNeighbor,entity)
3. **return** bestNeighbor

HANDLE-SOLUTION(*entity*)
1. UPDATE-BEST-SOLUTION(*entity*)
```

Fig. 3. BCO implementation.



a. Average execution time of BCO.



b. Comparison of ACS and BCO.

Fig. 4. Experiments with BCO and comparison with ACS.

entity that contains also its solution path. Therefore, while ACS uses by default the retracing method of agent notification when a better solution is found, the BCO uses the broadcast method. For BCO, method UPDATE-BEST-SOLUTION() simply saves the best known solution. In contrast, ACS needs to mark the solution with pheromone, thus consuming more execution time (as required by σ ant moves).

6. Experimental Results

We tested the scalability of the framework on the HPC Infragrid cluster composed of machines using Intel Xeon CPU E5504 consisting of 8 cores. The computers are connected by an Infiniband 40 Gbits/s network. The cluster is running Linux operating system and Condor workload management system [14]. Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Each computer core was used as a separate computing node with 1GB of RAM.

The BCO parameters were set to the values recommended by [4], while the ACS parameters were set to the values recommended by [3]. The algorithms were applied to solve the TSP problem on the fully connected graph $pr1002$ with 1002 vertices, as defined in the benchmark maps from [20]. The stop condition was checked for the reference vertex 0, i.e. the first vertex in $pr1002$ input file. The condition checked that the reference vertex was visited by a number of $M = 10000$ entities.

We ran 9 experiments with BCO, each one consisting of $r = 10$ experimental rounds, on an increasing number of computing nodes varying from 1 to 60.

Fig. 4a clearly shows a decreasing trend for T_{avg} when the number of computing nodes is increasing. However, there are small exceptions such as the experiments on 20, 50 and 60 computing nodes where T_{avg} is higher than the predecessor experiment for less computing nodes (i.e. 10, 40 and respectively 50 nodes). The computing nodes from the cluster are organized into groups of 8 nodes per each physical machine (or server) of the cluster. So,

Table 2. Experiments with BCO

nodes	$T_{avg}[s]$	v_{avg}	$t_{avg}[s]$	v_{min}	$t_{min}[s]$	S_c
1	15135	319368	7045	319368	7045	1
2	7995	313146	4570	306392	5012	1.89
5	3118	297374	2106	291192	2464	4.85
10	1523	310020	7729	306768	1228	9.93
20	1653	310848	1037	305844	1653	9.15
30	1245	313280	1064	281452	1294	12.15
40	458	313588	454	309463	432	33.04
50	791	310224	681	306596	703	19.13
60	798	313768	622	310598	713	18.96

Table 3. BCO compared with ACS

SI alg.	nodes	$T_{avg}[s]$	v_{avg}	$t_{avg}[s]$	v_{min}	$t_{min}[s]$
BCO	1	15135	319368	7045	319368	7045
ACS	1	22718	321520	3943	310220	2889.6
BCO	5	3118	297374	2106	291192	2464
ACS	5	4844	308027	727	307004	841.9
BCO	10	1523	310020	7729	306768	1228
ACS	10	2312	307848	252	306768	281.7
BCO	20	1653	310848	1037	305844	1653
ACS	20	1536	307036	308	305216	395.9

experiments with 10 computing nodes are using just 2 physical machines, i.e. two 8-processor servers, while an experiment with 20 computing nodes requires 3 physical machines and implicitly more network bandwidth of the Infragrid connection. In the case of the experiment with 40 computing nodes, there are used all the 8 processors from each of the 5 involved servers. In contrast, for the 30 node experiment, 4 servers were used. Each of the first 3 servers used 8 nodes, while the last server used only 6 nodes. Therefore, a higher volume of network communication occurred for the last server.

Our experiments show that the best performance for a graph of 1002 vertices was obtained when we used 40 computing nodes. This is also indicated by the last two rows of Table 2. It presents a final increase of T_{avg} although the number of computing nodes was also increased. We obtained a decrease of T_{avg} from $T_{avg} = 15135s$ when using 40 computing nodes to $T_{avg} = 458s$ when using a single computing node. However the best speedup was obtained for 10 computing nodes resulting in an efficiency of 0.99 which surpasses the best efficiency value of 0.9 that we have encountered in our literature review.

From this discussion, we can safely conclude that BCO implemented using our architecture is indeed scalable.

The comparison of the BCO and ACS implementations presented in this paper gives an idea of which SI algorithm performs better on our architecture. Both SI algorithms were tested on the same TSP map pr1002 of 1002 vertices from the benchmark library TSPLIB using a variable number of computing nodes $k \in \{1, 5, 10, 20\}$. Table 3 presents the performance values obtained by experimenting with ACS and the results obtained by BCO. Fig. 4b is a graphical representation of the data from Table 3. that clearly shows that BCO is actually faster than ACS. This is due to the fact that the BCO process of broadcasting the best solution is faster. BCO only needs to save the best known entity which is an operation of $O(1)$ complexity and does not need to mark a pheromone trail of size n which is an operation of $O(n)$ computational complexity (the case of ACS). Our experiments show that our framework peaks at 15 computing nodes using ACS, respectively at 10 computing nodes when using BCO. See paper [2] for details on scalability of ACS implemented on our framework.

7. Conclusions and Future Work

This paper presented a new framework based on our previous work ACODA, for distributed implementation of SI algorithms aimed at solving complex graph search problems. Examples of implementations and experimental results were given for SI algorithms of ACS and BCO for solving the TSP.The conclusion being that our approach scales better when implementing the ACS algorithm but is faster when implementing BCO.

As future work we plan to thoroughly compare our framework with "island approach" for the distribution SI algorithms. For this purpose we must first implement this approach using distributed multi-agent middleware in order to obtain a fair comparison. We also plan to extend the peak efficiency of the architecture by balancing the load of the agents at runtime in order to minimize network communication. Our recent simulation results reported in [26] show that the communication overhead could be reduced up to 50%.

Acknowledgements

We thank the members of HPC Center of West University of Timişoara, Romania[1], for allowing us to run our experiments on their high-performance computing infrastructure.

References

[1] A. P. Englebrecht, Fundamentals of computational swarm intelligence, Wiley, 2005.

[2] S. Ilie, C. Bădică, Multi-agent approach to distributed ant colony optimization, Science of Computer Programming(In press, corrected proof. Accepted 15 September 2011, Available online 28 September 2011).
URL `http://dx.doi.org/10.1016/j.scico.2011.09.001`

[3] M. Dorigo, T. Stutzle, Ant Colony Optimization, MIT Press, 2004.

[4] L.-P. Wong, M. Y. H. Low, C. S. Chong, Bee colony optimization with local search for travelling salesman problem, International Journal on Artificial Intelligence Tools 19 (3) (2010) 305–334.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms (third edition), MIT Press, 2009.

[6] D. L. Applegate, R. E. Bixby, V. Chvtal, W. J. Cook., The Traveling Salesman Problem: A Computational Study, Princeton Series in Applied Mathematics, Princeton University Press, 2006.

[7] S. N. Parragh, K. F. Doerner, R. F. Hartl, A survey on pickup and delivery problems, Journal für Betriebswirtschaft 58 (2) (2008) 755–762.

[8] F. C. Dyer, The biology of the dance language, Annual Review of Entomology 47 (1) (2002) 917–949.

[9] J. C. Biesmeijer, T. D. Seeley, The use of waggle dance information by honey bees throughout their foraging careers, Behavioral Ecology and Sociobiology, 58 (1) (2005) 133–142.

[10] S. A. Subbotina, A. A. Oleinik, Multiagent optimization based on the bee-colony method, Cybernetics and Systems Analysis 45 (2) (2009) 177–186.

[11] D. Teodorovic, Bee colony optimization (bco), in: Innovations in Swarm Inteligence, Vol. 248 of Studies in Computational Intelligence, Springer, 2009, pp. 39–60.

[12] H. Hoos, T. Stutzle, Stochastic Local Search: Foundations and Applications, Morgan Kaufmann, 2005.

[13] N. Santoro, Design and Analysis of Distributed Algorithms, Wiley, 2006.

[14] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the condor experience, Concurrency and Computation: Practice and Experience 17 (2–4) (2005) 323–356.

[15] M. Pedemonte, S. Nesmachnow, H. Cancela, A survey on parallel ant colony optimization, Applied Soft Computing 11 (8) (2011) 5181–5197.

[16] M. Pedemonte, H. Cancela, A cellular ant colony optimization for the generalized steiner problem, International Journal of Innovative Computing and Appllications 2 (3) (2010) 188–201.

[17] R. S. Parpinelli, C. M. V. Benitez, H. S. Lopes, Parallel approaches for the artificial bee colony algorithm, in: Handbook of Swarm Intelligence: Concepts, Principles and Applications, Adaptation, Learning, and Optimization, Springer, 2010, pp. 329–345.

[18] M. Subotic, M. Tuba, N. Stanarevic, Different approaches in parallelization of the artificial bee colony algorithm, International Journal of Mathematical Models and Methods in Applied Sciences 5 (4) (2011) 755–762.

[19] J. Chintalapati, M. Arvind, S. Priyanka, N. Mangala, J. Valadi, Parallel ant-miner (pam) on high performance clusters, in: Swarm, Evolutionary, and Memetic Computing, Vol. 6466 of Lecture Notes on Computer Science, Springer, 2010, pp. 270–277.

[20] G. Reinelt, Tsplib – a traveling salesman library, ORSA Journal on Computing 3 (4) (1991) 376–384.

[21] L. Chen, H.-Y. Sun, S. Wang, Parallel implementation of ant colony optimization on mpp, in: Proc. International Conference on Machine Learning and Cybernetics – ICMLC'2008, Vol. 2, IEEE, 2008, pp. 981–986.

[22] A. Banharnsakun, T. Achalakul, B. Sirinaovakul, Artificial bee colony algorithm on distributed environments, in: Second World Congress on Nature and Biologically Inspired Computing (NaBIC'2010), 2010, pp. 13–18.

[23] T. Holvoet, D. Weyns, P. Valckenaers, Patterns of delegate mas, in: Proc. of the 3rd IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO'09, IEEE, 2009, pp. 1–9.

[24] H. Wedde, M. Farooq, Y. Zhang, Beehive: An efficient fault-tolerant routing algorithm inspired by honey bee behavior, in: M. Dorigo, M. Birattari, C. Blum, L. M. Gambardella, F. Mondada, T. Stützle (Eds.), Proc.4th International Workshop of Ant Colony Optimization and Swarm Intelligence (ANTS'2004), Vol. 3172 of Lecture Notes in Computer Science, Springer, 2004, pp. 83–94.

[25] F. L. Bellifemine, G. Caire, D. Greenwood, Developing Multi-Agent Systems with JADE, John Wiley & Sons Ltd, 2007.

[26] C. Bădică, S. Ilie, M. Ivanovic, Optimizing communication costs in acoda using simulated annealing: Initial experiments, in: N. T. Nguyen, K. Hoang, P. Jedrzejowicz (Eds.), Proc.4th International Conference o Computational Collective Intelligence. Technologies and Applications, ICCCI 2012 (1), Vol. 7653 of Lecture Notes in Computer Science, Springer, 2012, pp. 298–307.

[1]http://hpc.uvt.ro