Transparent and Efficient Parallelization of Swarm Algorithms

FRANCO CICIRELLI, AGOSTINO FORESTIERO, ANDREA GIORDANO,
and CARLO MASTROIANNI, CNR Institute for High Performance Computing and Networks

This article presents an approach for the efficient and transparent parallelization of a large class of swarm algorithms, specifically those where the multiagent paradigm is used to implement the functionalities of bioinspired entities, such as ants and birds. Parallelization is achieved by partitioning the space on which agents operate onto multiple regions and assigning each region to a different computing node. Data consistency and conflict issues, which can arise when several agents concurrently access shared data, are handled using a purposely developed notion of logical time. This approach enables a transparent porting onto parallel/distributed architectures, as the developer is only in charge of defining the behavior of the agents, without having to cope with issues related to parallel programming and performance optimization. The approach has been evaluated for a very popular swarm algorithm, the ant-based spatial clustering and sorting of items, and results show good performance and scalability.

Categories and Subject Descriptors: D.1.3 [**Concurrent Programming**]: Parallel Programming; I.2.11 [**Distributed Artificial Intelligence**]: Multiagent Systems

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Swarm algorithms, logical time, ant-based clustering and sorting

ACM Reference Format:
Franco Cicirelli, Agostino Forestiero, Andrea Giordano, and Carlo Mastroianni. 2016. Transparent and efficient parallelization of swarm algorithms. ACM Trans. Auton. Adapt. Syst. 11, 2, Article 14 (May 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2897373

1. INTRODUCTION

Many biological and artificial systems exploit the swarm intelligence paradigm [Bonabeau et al. 1999, 2000]: several small and autonomous entities perform very simple operations driven by local information, and a complex and intelligent behavior emerges from the combination of such operations. For example, the ant foraging behavior investigated in Deneubourg et al. [1990a], which later inspired the ant colony optimization algorithms [Dorigo and Stützle 2004], allows some species of ants to establish the shortest path toward a food source. Such behavior emerges from the combination of the individual operations of ants, which while searching for food follow a pheromone substance deposited by other ants that have already discovered a food source. As another example, the flocking behavior described in Reynolds [1987] allows birds to travel in large flocks and rapidly adapt their movements to the changing characteristics of

This work was partially supported by MIUR-PON under project PON03PE_00050_2 within the framework of the Technological District on Smart Homes (DOMUS).
Authors' address: F. Cicirelli, A. Forestiero, A. Giordano, and C. Mastroianni, CNR Institute for High Performance Computing and Networks, Via P. Bucci 41C, 87036 Rende (CS), Italy; emails: {cicirelli, forestiero, giordano, mastroianni}@icar.cnr.it.
Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 1556-4665/2016/05-ART14 $15.00
DOI: http://dx.doi.org/10.1145/2897373

the environment. This behavior can be obtained by defining a set of simple rules that are followed by individual birds. Other examples observed in nature are animal herding, bacterial growth, and fish schooling. As an example of artificial system, swarm robotics is an approach to the coordination of multirobot systems that allows a desired collective behavior to emerge from the interactions among a large number of physical robots [Brambilla et al. 2013; Beni 2005; Dorigo et al. 2013].

Swarm intelligence algorithms, or briefly swarm algorithms, are heuristics that aim to solve complex problems by imitating the swarm behaviors observed in biological systems. Systems that rely on swarm intelligence exhibit several beneficial properties, such as the following: (1) self-organization, as decisions of individuals are based on local information (i.e., without any central coordinator), and (2) adaptivity, as individuals can react flexibly to the ever-changing environment. The individuals of a swarm can communicate with one another directly, or they can interact and cooperate through the modifications of the environment, such as using a communication modality referred to as stigmergy [Grassé 1959]. Examples of swarm algorithms include ant-based clustering and sorting [Deneubourg et al. 1990b], particle swarm optimization (PSO) [Kennedy and Eberhart 1995], ant colony optimization (ACO) [Dorigo and Stützle 2004], flock algorithms [Reynolds 1987], and bee algorithms [Karaboga and Akay 2009]. Swarm algorithms proved capable of solving complex tasks, such as coordinated decision making, task allocation, routing problems, and graph partitioning [Bonabeau et al. 1999; Navlakha and Bar-Joseph 2015]. Swarm intelligence systems can be modeled and implemented by exploiting the paradigm of agent-based computing [Wooldridge 2002; Sycara 1998; Ferber 1999]: in such a case, the individuals of a swarm are modeled by software agents.

The complexity and size of the problems tackled by swarm algorithms, and the large amount of involved data, often require resorting to the parallel/distributed execution of the algorithms. When porting an algorithm to a parallel/distributed architecture, it is necessary to provide safe and correct management of shared data. In many cases, the individuals operate in a territory. For example, in the case of ant-based clustering, the goal of the individuals is to spatially cluster a set of items by moving them over a bidimensional space, as better detailed in Section 2. In such cases, the territory itself is a huge shared variable that requires data consistency issues to be addressed. Concurrency issues are usually regulated through the use design and implementation of synchronization primitives, for example, to let threads acquire and release locks on shared data. However, the use of these primitives can lead to significant performance degradation and poor scalability, especially when the access to territory information is frequent and performed by a large number of entities [Pinciroli et al. 2012; Bueso 2015].

In this article, we present an approach and a software architecture aimed to the efficient parallelization of the swarm algorithms for which individuals can be implemented as agents embedded in a bidimensional territory. Our methodology exploits the space partitioning approach, which consists of assigning different regions of the territory to different computing servers. Data consistency issues are not tackled by resorting to lock-based mechanisms and high-level synchronization primitives, which can impair the execution performance and scalability. Conversely, our approach exploits a special-purpose notion of logical time [Lamport 1978] to manage shared information properly. Labels, such as natural numbers associated with logical time, are assigned to agents to enforce an order to their operations and prevent any concurrent write-mode access to shared data. Moreover, our approach offers an additional and significant benefit, in that the management of the shared territory is transparent for the developer, who remains in charge only of defining the behavior of the agents, without having to cope with issues related to parallel/distributed programming and performance optimization.

The remainder of the article is organized as follows. Section 2 discusses the ant-based clustering and sorting algorithm chosen as a testbed to show the effectiveness of the approach. Section 3 describes how the shared territory is partitioned to parallelize swarm algorithms and fasten their execution. Section 4 describes the mechanism, based on logical time, adopted to assign labels to agents and ensure a safe execution order while also discussing the effect of the labeling assignment on execution performances. Section 5 illustrates the software infrastructure that implements the approach, describes its benefits in terms of simplicity and transparency, and discusses how the architecture can be used for different types of swarm algorithms. Section 6 shows the performance of the parallel execution and reports a speedup analysis when varying the problem size, the number of parallel nodes, and the labeling patterns. Section 7 illustrates related work, and Section 8 concludes the article and provides indications about future research avenues.

2. THE ANT-BASED CLUSTERING AND SORTING ALGORITHM

The approach and the software architecture presented in this article can be used to parallelize a large class of swarm algorithms, ranging from bird flocks to PSO to bee-based algorithms, and so forth. For demonstration purposes, we focus on the ant-based clustering and sorting algorithm, inspired by the behavior of some species of ants that cluster corpses to form a “cemetery” or sort their larvae into separate piles. The basic characteristic features of the clustering and sorting behavior of ants can be reproduced by a simple model, first presented in Deneubourg et al. [1990b] and later extended in Bonabeau et al. [1999], in which agents move over a territory and pick up and deposit items on the base of local information.

If items are all of the same kind, the goal of ant-based clustering is to create regions in which items are accumulated, leaving empty regions in between. If items belong to several different types, or classes, the objective becomes to sort items spatially (i.e., separate items of different classes and cluster items of the same class). In the following, we refer to the sorting model (i.e., to the case in which items belong to different classes), since the clustering model can be considered as a special case of the sorting model.

The territory is modeled as a bidimensional space organized in a grid of cells. Each agent has visibility over the items located in its own cell and in the cells distant no more than R_V cells. R_V is the *visibility radius*, and the set of cells defined in this way is referred to as the visibility area of the agent. Each agent contributes to the spatial sorting of items by picking and dropping items from/to the cells. The agents perform their operations by following a timestep advancement schema. At each timestep, every agent moves randomly in the environment, toward an adjacent cell, and in the new cell performs a drop or pick attempt, according to whether it already holds an item, picked from another cell, or not. The *pick* and *drop* operations are driven by corresponding probability functions.

Let C be the number of predefined classes, and let $c = 1...C$ be the class of a given item. The probability with which an agent picks an item of a given class c from the cell where it is currently located, referred to as $P_{pick}(c)$, is defined in formula (1):

$$P_{pick}(c) = \left(\frac{k_p}{k_p + f(c)}\right)^2. \tag{1}$$

In formula (1), $f(c)$ gives the number of items of class c, accumulated in the cells within the visibility area of the agent, divided by the overall number of items of *all* classes that are located in the same area. As more items of a class c are accumulated in the visibility area of the agent, $f(c)$ increases and the value of the pick probability for this class becomes lower, and vice versa. This has the effect of inducing agents to

pick items that are uncommon in the visibility area and to ignore items of the class that is being accumulated. The parameter k_p is a nonnegative value used to tune the clustering effort. In the tests performed in this work, it is set to 0.1, as in Deneubourg et al. [1990b].

The probability that a loaded agent drops an item of class c, $P_{drop}(c)$, is defined in formula (2):

$$P_{drop}(c) = \left(\frac{f(c)}{k_d + f(c)} \right)^2. \tag{2}$$

The drop probability increases as more items of class c are accumulated in the visibility area of the agent. In this work, the parameter k_d is set to 0.3, as in Deneubourg et al. [1990b].

The effectiveness of the sorting algorithm can be evaluated through a spatial entropy function, based on the well-known Shannon formula for the calculation of information content. For each cell l, the local entropy $E(l)$, defined in formula (3), gives an estimation of the extent to which the items have been spatially sorted in the local area of the cell l, defined as the area that includes the cell l and its adjacent cells. In formula (3), $g(c, l)$ is the fraction of items of class c that are located in the local area of the cell l with respect to the overall number of items located in the same area:

$$E(l) = \frac{\sum_{(c=1\ldots C)} g(c,l) \cdot \lg \frac{1}{g(c,l)}}{\lg C}. \tag{3}$$

The function $E(l)$ is normalized so that its value is comprised between 0 and 1. In particular, an entropy value equal to 1 corresponds to the presence of comparable numbers of items of the different classes, whereas a low entropy is obtained when the local area has accumulated a large number of items belonging to one specific class. The overall entropy E is defined as the average of the entropy values $E(l)$ computed at all system cells. The overall entropy measures how well items are sorted in the territory.

3. PARTITIONING THE TERRITORY FOR PARALLELIZING SWARM INTELLIGENCE ALGORITHMS

As the problem size increases, it is convenient to parallelize or distribute the execution of swarm algorithms [Pedemonte et al. 2011; Yang et al. 2012; Twomey et al. 2010]. As stated in Section 1, we focus on the cases that individuals can be modeled as agents that own spatial coordinates and are embedded into the territory (spatial environment) where they move and live [Wooldridge 2002; Ferber 1999]. We assume that the swarm algorithm is implemented in a step-based fashion—that is, at each timestep, all agents perform their tasks before passing to the next timestep—and that the territory is managed as a bidimensional grid of cells. Cells can contain agents, which can move from one cell to another, and objects. The notion of *visibility radius* R_V, introduced in Section 2, is exploited to delimit the visibility area within which an agent is able to perceive the surrounding space. Analogously, the *action radius* R_A is defined as the distance, expressed in number of cells, between the cell where an agent resides and the farthest cell on which that agent is able to perform modifications, and delimits the action area.

In a parallel/distributed scenario, the territory is a huge shared variable of a concurrent system. A recurrent access of agents to the territory can easily become a bottleneck that limits system performance and scalability. A solution is to partition the territory, and its content (objects and agents), among multiple computing nodes. The territory is split into equal-sized regions, as shown in Figure 1. Each region is allocated to a different computing node [Cicirelli et al. 2007, 2015]. The agents located in a region

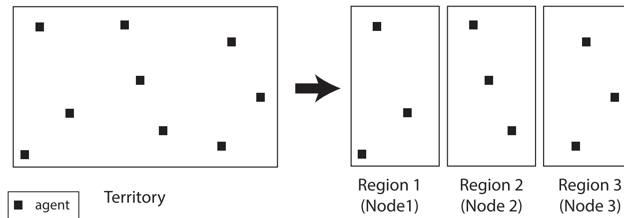


Fig. 1. The territory is split into regions that are associated with parallel computing nodes.

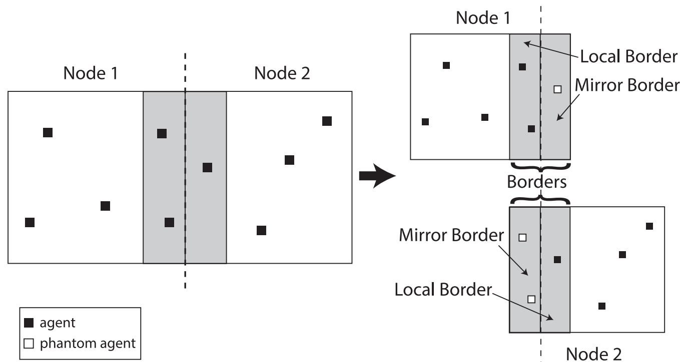
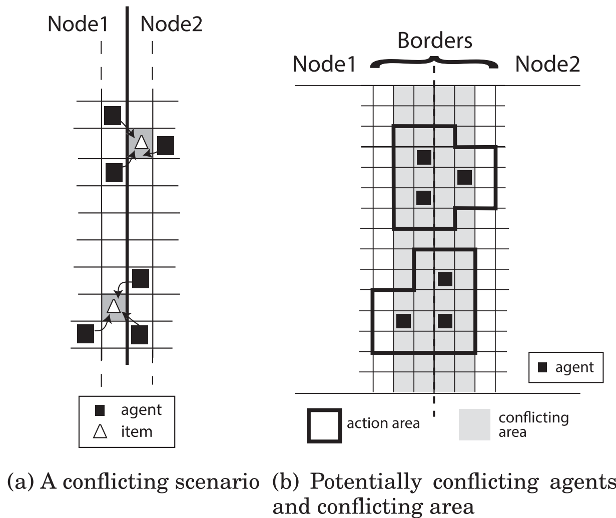


Fig. 2. Border areas of two adjacent nodes.

are executed by the corresponding node. Each node operates sequentially—that is, a nonpreemptive interleaved execution of agent actions is adopted. This partitioning reduces the amount of shared data and attenuates consistency and conflict resolution issues. Indeed, two agents that are located in different regions in most cases access different pieces of data. However, it can still happen that agents execute concurrently and access the same data. This occurs when the visibility and/or the action area of an agent extend beyond the boundary of the local region and include the border of an adjacent region. In such cases, the agent can operate on data that is concurrently accessed by one or more agents of the adjacent region. This event generates a conflict situation that can lead to data inconsistency.

Usually, conflict resolution and consistency are achieved by resorting to synchronization primitives (e.g., locks), which, however, can present two main drawbacks: (1) they can hinder the transparency of the parallelization procedure, as the developer is compelled to cope with the management of such primitives, and (2) they can negatively affect performance and scalability. Our approach allows the mentioned issues to be tackled by using a methodology, based on logical time [Cicirelli et al. 2014], which is able to transparently enforce a conflict-free and fair execution order on concurrent actions. This methodology is detailed in Section 4.

To reduce internode communications and improve performance, our approach ensures that each agent operates only on objects hosted by the same computing node where the agent resides. This is achieved by replicating the edge portion of a region in adjacent nodes. This edge portion is referred to as a border of the region, as shown in Figure 2. The border area of a given region is made up of two distinct parts: the local border and the mirror border. The first is managed by the local node, and information updates are replicated in the mirror border of the adjacent node. For example, information about the updates occurred in the local border of Node 1 of Figure 2 are sent to Node 2, which applies the updates in its mirror border. Analogously, information in



(a) A conflicting scenario (b) Potentially conflicting agents and conflicting area

Fig. 3. Scenario with agents conflicting on the borders of two nodes. In this example, $R_A = 1$.

the mirror border of Node 1 is aligned with the updates occurring in the local border of Node 2. Agents located in a border area are mirrored by means of *phantom* agents (copies of the original agents that are not actually executed), whereas items are simply duplicated. Information about the updates is exchanged by means of update messages: at the end of each timestep, every computing node sends a single message to the adjacent nodes containing information about all updates that have occurred in the local border area during the last timestep. More details about the management of update messages are given in Section 5.

4. USE OF LOGICAL TIME FOR CONFLICT-FREE PARALLEL EXECUTION

As mentioned in the previous section, splitting the territory and spreading the agents over different computing nodes can raise data consistency issues. This is clarified in Figure 3(a), which shows an example of a conflicting scenario where some agents compete to pick the same items contained in the grey cells. As the agents operate concurrently, two or more of them can try to pick the same item: if pick operations are actually performed, this will lead to an inconsistent state of the algorithm. As stated in the previous section, each node operates sequentially, and agents can execute concurrently only if they are located on different regions. As shown in Figure 3(b), two agents are potentially conflicting when they belong to different regions and their action areas overlap—that is, they are separated by at most $2R_A - 1$ interposed cells. In the example of Figure 3(b), $R_A = 1$, so agents are potentially conflicting if the number of interposed cells is equal to 1.

To prevent conflicts, we borrow the notion of logical time from the distributed systems field. The logical time concept [Lamport 1978] is typically used to prevent causality-constraint violations in distributed systems. In our case, logical time is used to establish a partial order of agent executions during a given timestep such that potentially conflicting agents cannot execute concurrently. The logical time notion as a tie-breaking mechanism for preventing conflicts was first used in Cicirelli et al. [2015] in the context of event-driven distributed simulation. Here the approach is refined, exploited, and evaluated for the distributed execution of swarm algorithms. More specifically, at every timestep, labels (natural numbers) are assigned to agents so that potentially

conflicting agents are assigned different labels. The labels are used as a logical time that enforces a conflict-free execution order. At each timestep, every computing node executes the agents located in its corresponding region, respecting the label ordering: it executes all agents with label 1, then those with label 2, and so forth. To ensure algorithm consistency during parallel execution, the nodes must synchronize among themselves. When a node completes the execution of all agents with a given label, it notifies this to its adjacent nodes by sending them a completion message[1] and proceeds to the next label after receiving analogous messages from them. This ensures that two adjacent nodes cannot concurrently execute agent operations associated with different labels.

Before providing details on the labeling mechanism, some definitions are needed. Let us consider two cells, c_1 and c_2, belonging to two adjacent regions and separated by at most $2R_A - 1$ interposed cells. Every two agents that are located, one in c_1 and the other in c_2, are potentially conflicting, in accordance to the definition given previously. Such two cells are called *potentially conflicting* cells. We then define a *conflicting area* $A_C(i, j)$ of two adjacent regions R_i, R_j as the portion of the border region that includes all couples of potentially conflicting cells (see the gray part of Figure 3(b)).

An easy fashion to perform a conflict-free labeling of agents is to first assign labels to the cells belonging to the conflicting areas and then assign each agent the label of the cell where it is located. Any mechanism for the assignment of labels to cells should provide that conflicting agents are never executed concurrently. This property, referred to as conflict-free property in the following, is guaranteed if every two potentially conflicting cells are assigned different labels.

4.1. Labeling the Territory: Pattern and Schema

For the sake of clearness, we distinguish between a pattern and a schema: the former determines how to assign labels to the cells of a conflicting area, whereas the latter determines the assignment of labels to the whole territory. If the x and y coordinates of the cells of a conflicting area are expressed starting from the left-top cell, a *pattern* $P(x, y)$ is defined as a function $P : A \rightarrow L$ that associates a set of spatial coordinates A (where $A = \{(x, y) \in \mathbb{N}^2 : 0 \leq x < D_x \wedge 0 \leq y < D_y\}$ and D_x, D_y are respectively the x and y sizes of the conflicting area) with a set of label L. The labeling generated by a pattern must satisfy the conflict-free property. As an example, the pattern depicted in Figure 4, referred to as Pattern P_A, satisfies the property in the case in which the action radius R_A is equal to 2, because the number of cells interposed between two cells located in different nodes and having the same label is always larger than $2R_A - 1$, which is this case equals 3. Pattern P_A assigns labels between 0 and 7 to cells, using the following expression:

$$P_A(x, y) = \left| \left\lfloor \frac{y\%16}{8} \right\rfloor - \left\lfloor \frac{x}{2} \right\rfloor \right| \times 4 + \left\lfloor \frac{y\%8}{4} \right\rfloor \times 2 + (x + y)\%2. \quad (4)$$

More examples of pattern are discussed later.

A *schema* extends the pattern to the whole territory and is defined as a function $S : T \rightarrow L$, where $T = \{(x, y) \in \mathbb{N}^2 : 0 \leq x < T_x \wedge 0 \leq y < T_y\}$ and T_x and T_y are respectively the x and y sizes of the territory.

The adoption of a given labeling affects the execution time and can introduce some biasing execution constraints. Before discussing these aspects—in Sections 4.2 and 4.3, respectively—we first introduce two strategies that can be used to combine the pattern and the schema. The first, referred to as naive strategy, consists of using a

[1]The update messages, used to update information on border regions (see Section 3), have the role of completion messages as well.

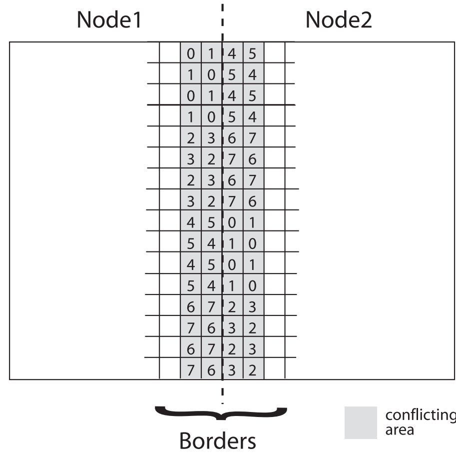


Fig. 4. Pattern P_A adopted for assigning labels to cells in the conflicting area.

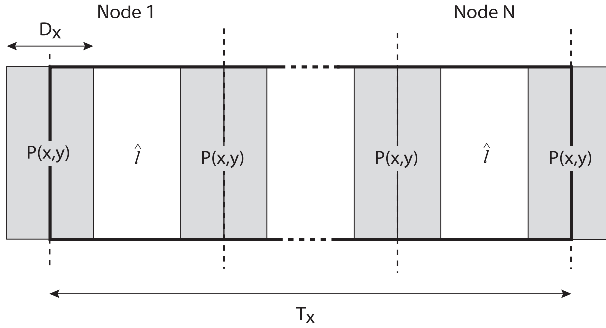


Fig. 5. Labeling the territory by adopting the naive strategy.

given pattern for all conflicting areas and assigning a fixed label to all other cells of the territory (see Figure 5). This strategy ensures the conflict-free property over the whole territory: indeed, the property is guaranteed by the pattern within the conflicting areas, whereas a single label is sufficient in the rest of the territory, because outside of the conflicting areas there cannot be potentially conflicting agents and therefore the execution order is irrelevant. Given any specific pattern $\bar{P}$, this strategy corresponds to using the schema S_{naive}:

$$S_{naive}(\bar{P}) = \begin{cases} \bar{P}\left(\left(x + \frac{D_x}{2}\right) \% \left(\frac{T_x}{N_R}\right), y\right) \text{ if } (\mathrm{x}, \mathrm{y}) \in \mathrm{A}^*_{\mathrm{C}} \\ \hat{l} \in L \quad \text{otherwise} \end{cases}, \tag{5}$$

where N_R is the number of regions, A^*_C is the union of all conflicting areas, and $\hat{l}$ is the fixed label assigned to the cells outside of the conflicting areas.

The second strategy, referred to as repetitive strategy, consists of replicating the labeling defined by a pattern $\bar{P}$ over the whole territory (see Figure 6). This is possible when T_x, the horizontal size of the territory, is a multiple of D_x, the horizontal size of a conflicting area. When adopting this strategy, the schema is defined as follows:

$$S_{repetitive}(\bar{P}) = \bar{P}\left(\left(x + \frac{D_x}{2}\right) \% (D_x), y\right). \tag{6}$$

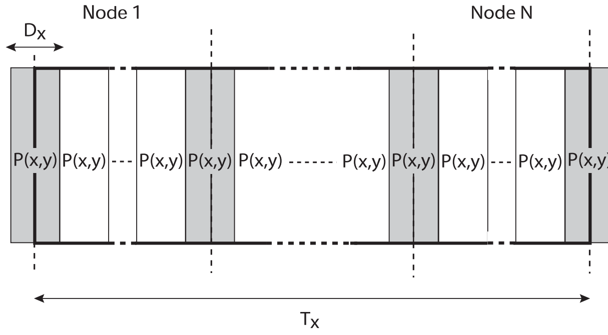


Fig. 6. Labeling the territory by adopting the repetitive strategy.

4.2. Cost of Synchronization

The time needed for synchronization is an overhead time, most of which is spent by a faster node to wait for an adjacent slower node before advancing the agents' execution to the next label. This overhead time increases both with the amount of load imbalance between adjacent nodes and the number of synchronization points. These two aspects are both affected by the labeling assignment, as illustrated in the following.

The load imbalance is minimized if the numbers of agents tagged with the same label in adjacent nodes are made as comparable as possible. We define the *label load* $LL_R(l, t)$ as the number of cells of the region R tagged with the label l at the timestep t. Of course, the number of agents tagged with the label l is related to $LL_R(l, t)$. As a particular case, the relationship is linear in the case in which the agents are uniformly spread over the territory. For this reason, the minimization of the load imbalance is related to minimizing $|LL_{R_x}(l, t) - LL_{R_y}(l, t)|$ for any two adjacent regions R_x and R_y. With both the basic and repetitive schemas described earlier, and depicted in Figures 5 and 6, the load imbalance is actually minimized. Indeed, the result is that the number of cells having a given label is the same for all computing nodes, regardless of the adopted pattern, and therefore $|LL_{R_x}(l, t) - LL_{R_y}(l, t)|$ is equal to 0 for any two regions R_x and R_y.

The number of synchronization points corresponds to the number of different labels assigned to cells, as within each timestep there is one synchronization point per label. Therefore, the cost of synchronization is affected by the labeling pattern. As an example, the pattern P_B, shown in Figure 7, uses a different label for each cell, thus generating a large number of synchronization points. The corresponding pattern function is defined in expression (7). Conversely, the pattern P_C, shown in Figure 8, requires only two synchronization points. The corresponding pattern function is reported in expression (8). Unfortunately, using a low number of labels can also have negative consequences, as described in the following section.

$$P_B(x, y) = x + (y \times D_x) \tag{7}$$

$$P_C(x, y) = \left\lfloor \frac{2x}{D_x} \right\rfloor \tag{8}$$

4.3. Execution Constraints

The adoption of a mechanism for conflict avoidance can affect algorithm evolution. The proposed labeling mechanism imposes a specific execution order not only among conflicting agents but also among nonconflicting ones. As a consequence, some evolutions of the algorithm, even correct and admissible, cannot occur. This can lead to two

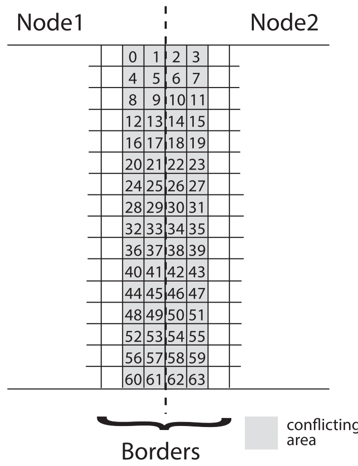


Fig. 7. Pattern P_B adopting a different label for each cell.

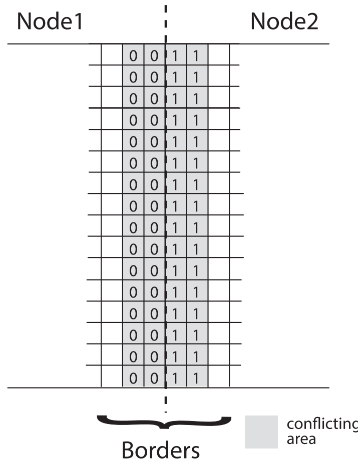


Fig. 8. Pattern P_C adopting only two labels.

phenomena that are illustrated in the following, referred to as biased execution and coarse-grain execution.

The phenomenon of biased execution can have an effect when agents compete for the acquisition of shared resources. This is due to the execution order imposed by the labeling mechanism: the agents with a lower value of the label are executed earlier than those with higher values and therefore have higher probabilities of gaining the control of a shared resource. An effective approach to solve this problem is to keep the same labeling schema along the whole algorithm execution but *shuffle* the label values, and therefore modify the agents' execution order, at every timestep.

Formally, let $F = f_0, \ldots, f_n$ be the set of all bijective functions $f_i : L \rightarrow L$, where L is the set of adopted label values. A permutation of the label values is achieved by applying the same function f_i to each label. A shuffling of a schema S is obtained by using expression (9) where, at each timestep, a different function f_i is randomly chosen. Within a timestep, the f_i function must be the same for all executing nodes; the burden of communicating the function among the nodes can be avoided by using, at each node,

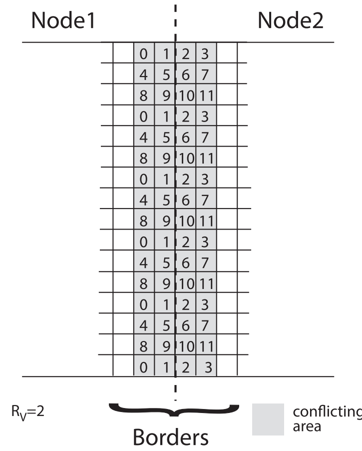


Fig. 9. Pattern P_D in the case in which $R_V = 2$.

the value of the timestep as a seed of the random generation process.

$$S_{shuffled}(x, y) = f_i(S(x, y)) \tag{9}$$

The second type of execution constraint is the coarse-grain execution, which is induced by the fact that all agents labeled with the same value must be executed consecutively, with no chance of interleaving the execution of agents with a different label. This introduces a granularity of agents' execution that is not related to the semantics of the application but is induced by the labeling mechanism itself. The effect of this phenomenon can be attenuated by reducing the number of cells having the same label, which corresponds to increasing the number of adopted label values. As an example, in the case of pattern P_C (see Figure 8), the coarse-grain execution phenomenon occurs and can have a nonnegligible effect. Conversely, when using the pattern P_B (see Figure 7), the phenomenon does not occur, because all cells have different labels, but the excessive number of labels can cause a remarkable cost of synchronization, as discussed in Section 4.2. The coarse-grain execution phenomenon can be avoided even with a lower number of labels than the one used by P_B. In practice, it is sufficient that any two agents that potentially interfere with each other (i.e., such that their distance is at most R_V cells) are labeled differently. The pattern P_D, depicted in Figure 9 and defined by expression (10), satisfies this property and can be used in place of the pattern P_B, with the advantage of using only $(R_V + 1) \cdot D_x$ different labels.

$$P_D(x, y) = x + (y\%(R_V + 1) \times D_x) \tag{10}$$

As a conclusive remark, it is observed that the number of different labels used in the assignment pattern has two opposite effects: a larger number of labels increases the number of synchronization points, possibly deteriorating the execution performance, but alleviates the problem of coarse-grain execution. Therefore, the assignment pattern should be chosen depending on the relative importance of these two aspects, which must be evaluated in any specific scenario. More specifically, execution constraints can be a problem when the assignment of a shared resource on one (class of) agent or another is important, such as when a score is assigned to each agent, or in pray-predator contexts. In such cases, the domain expert can be asked to evaluate the proper trade-off between the possible issues caused by execution constraints and the cost of synchronization. In other domains, execution constraints are not a problem, as in the case of the ant-based clustering and sorting algorithm, where all ants concur and cooperate to achieve the

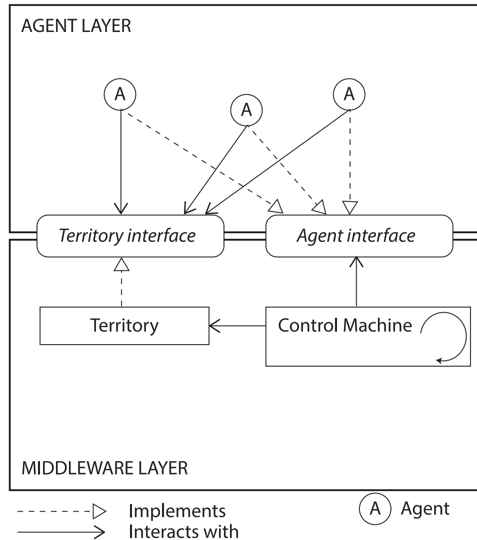


Fig. 10. Architecture in a sequential scenario.

same objective, with no regard on the partial performance of single agents or classes of agents. In such contexts, the labeling pattern can be chosen focusing only on the objective of minimizing the cost of synchronization.

5. IMPLEMENTATION

This section has three main goals: (1) to supply technical details to clarify the proposed methodology; (2) to show how our approach ensures that swarm algorithms can be transparently ported from a sequential to a parallel scenario, with no changes in the implementation code; and (iii) to give hints on how some well-known swarm algorithms can be implemented upon our infrastructure. In the following, we first describe the sequential execution scenario to introduce the basic components of our architecture. Then we analyze the parallel execution context and focus on the mechanisms involved in the transparent parallelization.

Our infrastructure, written in Java,[2] is composed of two coarse-grain architectural layers: the *Agent layer* and the *Middleware layer*. The former contains the agents that implement and execute the operations of swarm individuals. The latter implements the mechanisms and protocols described in the previous section and contains two main components: the *Territory component*, which copes with the management of the territory, and the *Control Machine*, which executes the main control loop through which the agent operations are executed step after step. Figure 10 focuses on the sequential scenario. The figure shows the two layers of the software architecture and the main involved components, and it suggests how the layers and the components interact with each other.

The two layers interact through two software interfaces: the Territory interface and the Agent interface. The *Territory interface* is used by agents to read/write the territory space regardless from the specific implementation of the Territory component. Analogously, the *Agent interface* is used by the Control Machine to execute the agent operations regardless of the specific nature of the agents and the implementation of their functionalities. This decoupled structure facilitates the transparent porting from

[2]The source code of the software is available at http://apswarm.icar.cnr.it.

Territory Interface
void **pickAgent**(Agent A, Location L) ▷ *picks the agent A hosted by the cell in location L*
void **dropAgent**(Agent A, Location L) ▷ *drops the agent A into the cell in location L*
void **moveAgent**(Agent A, Location OldL, Location NewL) ▷ *moves the agent A from OldL to NewL*
Agent[] **readAgents**(Location L) ▷ *reads all the agents located in the cell in location L*
void **pickObject**(Object O, Location L) ▷ *picks the object O hosted by the cell in location L*
void **dropObject**(Object O, Location L) ▷ *drops the object O into the cell in location L*
Object[] **readObjects**(Location L) ▷ *gets all the objects of the cell in location L*

Agent Interface
void **executeStep**() ▷ *execute agent code*

Fig. 11. The Territory and Agent interfaces.

the sequential to the parallel case. Indeed, the correct use of the Territory interface is the only requirement that an agent must satisfy to execute the code independently of the type of execution scenario, sequential or parallel. Once they respect this requirement, agents are free to define and use any data structure and perform all of the computation they need. All mechanisms and details related to the parallelization process, as described in this article, are managed behind the scenes by the Territory component and by the Control Machine, as will be better specified in the following.

The two mentioned interfaces are shown in Figure 11. The Agent interface only contains the *executeStep* method that must be implemented by any agent to define its behavior. The Territory interface consists of a set of methods that allow an agent to pick/drop another agent/object from/to a specific cell of the territory, to move itself or another agent from a cell to another, and to read data of objects/agents located in a specific cell. This interface contains the minimal set of methods needed to support the interactions between the agents and the territory in swarm algorithms. Obviously, the specific subset of required methods depends on the specific swarm algorithm. Some illustrative examples will be given at the end of this section.

Figure 12 shows the software architecture in the parallel execution context. The entire system is composed of a set of interconnected computing nodes or *servers*, which communicate among each other *a* through the exchange of messages. Each server has the same architecture as in the sequential scenario, except it includes one more component on the *Application layer*, the *Node Actor*, and one more interface, the *Node interface*, which concern the management of operations not assigned to agents. These two components will be described later. There is also one more component on the Middleware layer, the *Agent Label Manager*, which is in charge of dynamically managing the relationship between agents and labels. The Agent Label Manager assigns the labels to the cells at each timestep and ensures that the Control Machine executes the agent operations respecting the labels ordering (see Section 4). More specifically, the Control Machine iterates on labels and, for each label, asks the Agent Label Manager the set of agents tagged with the current label so as to execute their operations. To support this functionality, the Agent Label Manager is notified by the Territory about

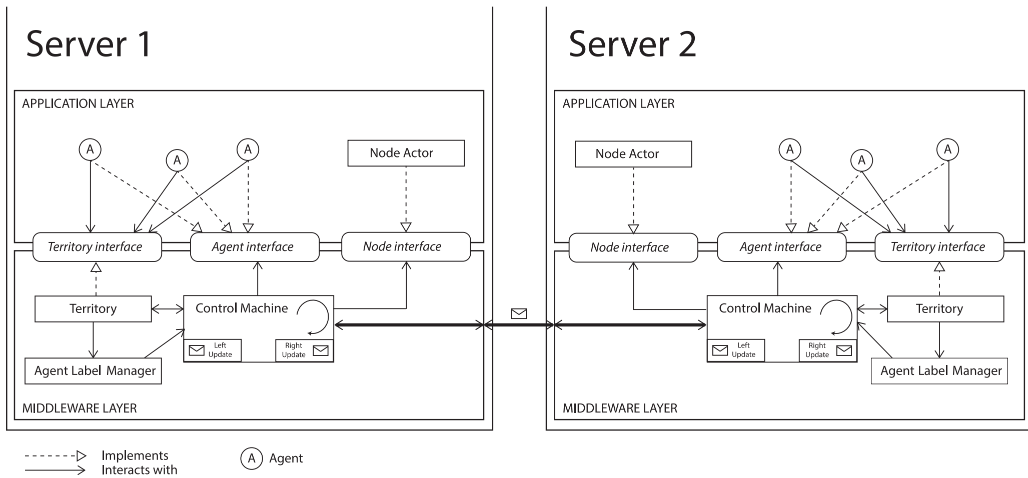


Fig. 12. Architecture in a parallel scenario with two servers.

any change in the locations of agents, as any agent movement induces a modification of its label. Figure 12 also shows that the Control Machine, besides being in charge of executing the main control loop, manages the update messages and the synchronization points. The update messages contain information about all of the changes occurring in the local borders that need to be sent to neighbor nodes. Left and right update messages respectively contain the update occurrences relative to the left and right local borders.

While in the sequential scenario the Territory component manages the entire territory space, in the parallel scenario it only manages the region assigned to the local server and the associated mirror borders. The Territory component is in charge of collecting the modifications occurring in the local borders. More specifically, when an agent uses one of the functionalities that modify the state of the environment (i.e., *pickAgent*, *dropAgent*, *moveAgent*, *pickObject*, *dropObject*), and the coordinates refer to a cell contained in the left/right local border, the Territory component transmits the relevant information to the Control Machine, which stores it respectively into the left/right update message that will be sent to the left/right neighbor server.

The mechanisms described previously require no changes in the agent code, as they are transparently managed by the Middleware layer. However, the agents still need to notify the Control Machine about the modifications of their own state, as these modifications are not detected by the Territory component. To preserve the main objective of the proposed architecture (i.e., the transparent porting from the sequential to the parallel execution context), this operation must be tackled without modifying the agent code. For this purpose, we adopt an aspect-oriented programming (AOP) technique [Kiczales et al. 2001]. AOP provides mechanisms to transparently change the source code in a dynamic fashion, at runtime, through the so-called Interceptor, a software entity included in the Middleware layer. In our case, when a modification occurs in the state of an agent located in the local border, the Interceptor captures the modified data and transmits it to the Control Machine. Hence, the Interceptor relieves the agent to cope with this aspect and allows the agent source code to not be altered.

Swarm algorithms can include the execution of operations not assigned to agents. Such operations are executed on all parallel nodes (e.g., the pheromone update in the case of ACO algorithms) and/or on a unique node, which takes the role of *Coordinator* and performs centralized operations (e.g., the computation of global metrics). In both cases, such operations are defined by the developer by supplying a Node Actor

Node Interface	
region_data **nodeStep**(int step)	▷ *executed at each step on each server*
void **globalStep**(region_data[])	▷ *executed at each step by the data manager*

Fig. 13. The Node interface.

component, which implements the Node interface, as depicted in Figure 12. The Node interface includes two methods, reported in Figure 13: the first, *nodeStep*, addresses the operations related to the local region and can collect information and produce modification upon all cells of the region. For example, if the swarm algorithm needs to manage pheromone trails (as in the ACO algorithms), the method can be used to execute the pheromone evaporation function on all cells of the region. The second method of the interface, *globalStep*, addresses operations related to the whole territory. The coordination of the two types of operations is ensured by the Control Machine that invokes the *nodeStep* method at each server and the *globalStep* method only at the Coordinator server. More precisely, the output of the *nodeStep* method of each server is sent to the Coordinator server, which collects all received data and uses it as input for the execution the *globalStep* method. As an example, let us consider the computation of the overall entropy defined in Section 2. The *nodeStep* method, executed at each server, computes the entropy value of the local region and returns it to the Control Machine, which sends the value to the Coordinator. The *globalStep* method, executed at the Coordinator server, analyzes the data received by the different servers and computes the overall entropy.

The pseudocode of the Control Machine loop is presented in Algorithm 1. The loop iterates on steps and, for each step, on labels. For a given step and label, the loop iterates on the agents tagged with that label. Each agent is triggered for execution by invoking its *executeStep* method (line 4). During the execution, the local border updates are properly stored in the left and right update messages. When the Control Machine terminates the execution of the agents associated with the current label, it executes the *sync* function (line 6), where the update messages are exchanged with *Control Machines* of the neighbors. To prevent deadlocks, first the Control Machine sends the messages to its neighbors, then it waits for the neighbors' messages (lines 12 through 15). After receiving the left and right messages, the Control Machine triggers the territory to update the information carried by these messages into the mirror borders of the local node (line 16). At the end of each step, the labels are shuffled (line 8) to avoid or at least reduce the biasing phenomenon described in Section 4.3. The *doStepBehaviour* function is called at the end of each step (line 9). In this function, the *nodeStep* of the Node Actor component is called and its output is sent to the Coordinator server (lines 19 and 20). The subsequent piece of code (lines 22 and 23) is executed only in the Coordinator server. Its objective is to wait for the data coming from the servers, collect such data, and then perform the *globalStep* method of the Node Actor component with that data as input. At this point, a new invocation of the *sync* function (line 25) ensures that all territory modifications are finalized before advancing to the next step.

The approach and the architecture presented in this article can be used to parallelize swarm intelligence algorithms transparently, specifically in the cases in which agents are embedded and operate in a bidimensional territory.[3] Any specific algorithm is modeled by defining the components of the Application layer (i.e., the Agents and the Node Actor) and the way in which these components interact with the territory through

[3]The architecture can be naturally extended to manage three or more dimensions. With the current version, however, a multidimensional territory must first be reduced to a bidimensional space through a dimension reduction technique.

ALGORITHM 1: Control Machine Loop

```
1: for each step s do
2:     for each label l do
3:         for each agent A having label l do
4:             A.executeStep();
5:         end for
6:         sync();
7:     end for
8:     shuffleLabels();
9:     doStepBehavior(s);
10: end for

11: function SYNC
12:     sendLeftUpdateMessage();
13:     sendRightUpdateMessage();
14:     msgL ← waitForLeftNeighborMessage();
15:     msgR ← waitForRightNeighborMessage();
16:     Territory.updateMirrorBorders(msgL, msgR);
17: end function

18: function DOSTEPBEHAVIOR(int step)
19:     thisRegionData ← NodeActor.nodeStep(step);
20:     sendToCoordinator(thisRegionData);
21:     if (this is the Coordinator) then
22:         regionData[] ← waitForRegionData();
23:         NodeActor.globalStep(regionData[])
24:     end if
25:     sync();
26: end function
```

the Territory interface. In the following, we give some details on how some well-known swarm algorithms can be implemented:

—*Ant-based clustering and sorting*: An agent of the ant-based algorithm described in Section 2 uses the *moveAgent* method of the Territory interface (Figure 11) to move itself across the territory and the *dropObject*/*pickObject* method to drop/pick an item to/from a cell. In addition, an agent explores its neighbor cells by using the *readObjects* method. Furthermore, the Node Actor implements the *nodeStep* and *globalStep* methods of the Node interface (Figure 13) to compute the local and overall entropy, as described previously.

—*Flocking*: The flocking behavior of birds, fish, or insects was modeled in Reynolds [1987] through agents (also called *boids*) that "fly" across a data space (e.g., to search interesting information). In our context, each agent moves to its next position (through the *moveAgent* method) after inspecting (through the *readAgents* method) the state of its neighbor agents, according to Reynolds' rules [Reynolds 1987]. Once in the new position, the agent uses the *readObjects* method to explore the local data and verify whether it has some desired properties. The *nodeStep* and *globalStep* methods of the Node Actor are used to collect and process the information discovered by agents respectively within each region and on the whole territory.

—*Particle swarm optimization*: Agents move in a territory corresponding to the search space of an optimization problem [Kennedy and Eberhart 1995]. In one of the most typical versions of the algorithm, the movements of each agent (performed through the *moveAgent* method) are influenced by the current best solution found by the agent itself, as well as by the current global best solution found by all agents. The

global best solution is stored and managed by the Node Actor and is updated each time an agent finds a better solution.

—*Ant colony optimization*: ACO algorithms are used to solve a large class of combinatorial optimization problems by taking inspiration from the foraging behavior of some species of ants [Dorigo and Di Caro 1999]. In this context, the parallelization that can be enabled by our methodology is of the kind proposed in Lin et al. [2007] (i.e., decompose the problem in subcomponents, with each subgraph assigned to a different node). In the case of the traveling salesman problem (TSP), this corresponds to assigning a subset of cities to each node. The current version of our infrastructure needs to be adapted, as ACO agents move across a graph structure instead of a bidimensional territory. Specifically, the Territory interface, shown in Figure 11, must be implemented so as to let a location identify a node of a graph rather than a cell of a bidimensional territory. Centralized operations of the ACO algorithm (i.e., pheromone update and local search) are performed through an ad hoc implementation of the Node Actor methods.

6. EXPERIMENTAL RESULTS

Performance evaluation was carried out for the ant-based sorting algorithm, used to spatially cluster items belonging to different classes, illustrated in Bonabeau et al. [1999], and summarized in Section 2. Our goal is twofold: (1) show that our methodology, while preventing data consistency issues, ensures a high degree of scalability in a wide set of scenarios, and (2) show the effect of the pattern (i.e., the assignment of labels to cells and agents in the conflicting areas (see Section 4)) on performance.

The adopted testbed is configured as follows. The number of items ranges between 60,000 and 600,000, and they are uniformly spread over a bidimensional grid of 240 $\times$ 100 cells. Items belong to a number of classes C between 3 and 9. The number of ant-like agents is proportional to the number of items and spans between 30,000 and 300,000. Each cell can be empty or contain one or more items and agents. The experiments were carried out on a cluster in which each computing node has and Intel Xeon E5-2670 CPU with 2.60GHz and 128GB RAM. The nodes are interconnected with an Intel Corporation I350 Gigabit Network.

To show how the ant algorithm clusters the items, Figure 14 reports three snapshots of the system taken before starting the algorithm, in an intermediate state, and when clustering has been achieved. The snapshots are taken for the scenario in which items belong to three classes and the visibility radius R_V is set to 10.

The value of the overall entropy, as defined at the end of Section 2, is computed every 1,000 steps and decreases as the algorithm proceeds, confirming its effectiveness. The system is considered stable when 10 successive values of the entropy differ among themselves no more than 1%: this is used as a stop criteria for the algorithm.

Before analyzing the parallel execution of the algorithm, we analyze its behavior when it is executed on a single node in some of the use cases of interest. Figures 15 and 16 show the behavior when varying the number of classes and the visibility radius, respectively. Specifically, Figure 15 shows that the entropy decreases from values close to 1 to values lower than 0.2, confirming that the items have been effectively clustered. The curves stop when the algorithm terminates its execution. It is noticed that as the number of classes C increases, the stable value of entropy decreases and the convergence of the algorithm is slightly faster. In these tests, the visibility radius R_V was set to 5. The value of R_V can be used to tune the size of the clusters, such as those of the areas containing items of the same class: larger clusters are obtained with larger values of R_V. Figure 16 shows the results of experiments in which the number of classes is set to 3 and the visibility radius ranges between 3 and 10. As the R_V value

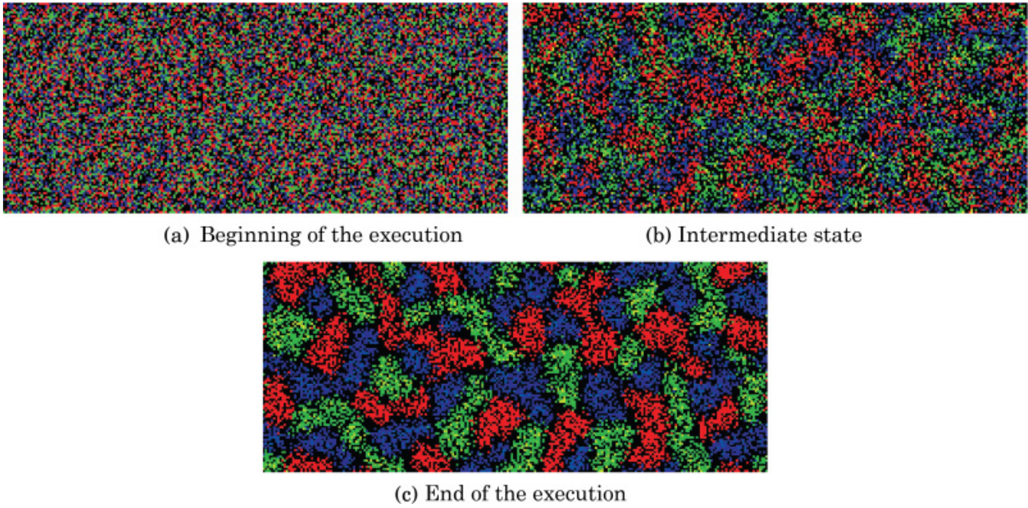


Fig. 14. Evolution of the ant-based sorting algorithm. The items belong to three classes, which correspond to the RGB colors. When a cell contains items of different classes, the color of the cell corresponds to the dominant class, and the color intensity is proportional to the number of items of the dominant class.

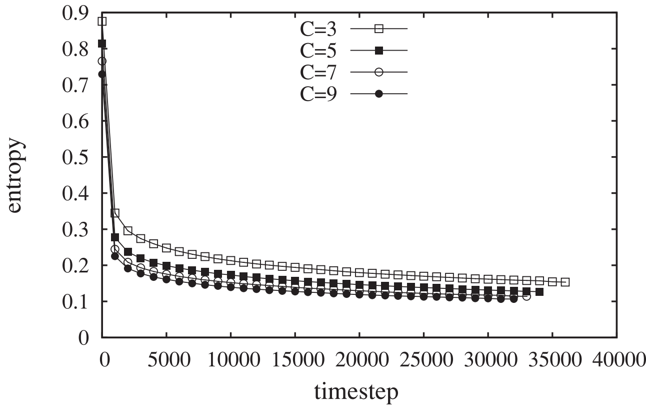


Fig. 15. Entropy curves using different values of C, the number of classes, with the visibility radius R_V set to 5. The curves stop when the algorithm terminates its execution.

increases, the algorithm needs more steps to converge, and it converges to larger values of entropy.

It should be remarked here that we obtained the same behavior as the one illustrated in Figures 15 and 16 both in the case of parallel execution and when the algorithm is executed outside our infrastructure. This mirrors the fact that the adopted approach does not alter the algorithm evolution.

In Section 4, we discussed the effect of the cell labeling pattern, in a parallel execution context, on the cost of synchronization and on the possible presence of execution constraints. We recall here that a lower number of different labels reduces the cost of synchronization but can induce some execution constraints (i.e., the coarse-grain execution phenomenon). As discussed in Section 4.3, in the case of the ant-based clustering and sorting algorithm, execution constraints are not a problem because all of the ants cooperate to achieve the same objective. As a confirmation of this, we found

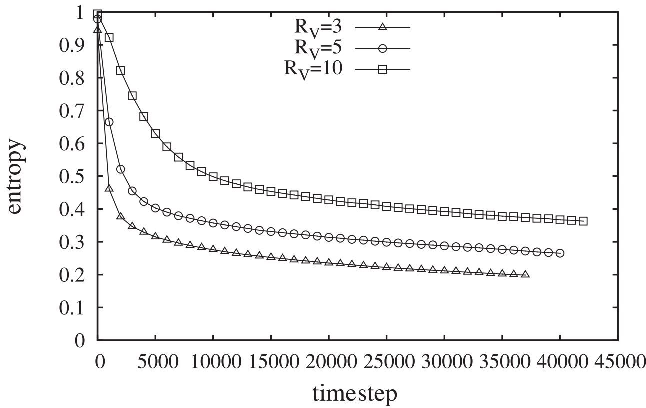


Fig. 16. Entropy curves using different values of R_V, the visibility radius, with the number of classes C set to 3. The curves stop when the algorithm terminates its execution.

experimentally that the trend of entropy values remains the same regardless of the adopted pattern. For this reason, we performed experiments adopting the pattern P_C, depicted in Figure 8, as this pattern uses the minimum number of different labels and ensures the best performance in terms of synchronization cost.

We performed two sets of experiments: in the first set, we investigated the performance when the problem size (i.e., the number of items) varies, and we ran the algorithm first on a single node and then on three parallel nodes of a cluster; in the second set, we performed a strong scalability evaluation[4] (i.e., for a fixed problem size, we varied the number of parallel nodes up to 9). In the considered scenario, the problem size increases with the number of items and the visibility radius. Indeed, the number of pick/drop attempts is proportional to the number of items, whereas the computational load of a single pick/drop attempt is proportional to the visibility radius, as the radius determines the number of cells involved in the computation of function $f(c)$ (see expressions (1) and (2) in Section 2). On the other hand, the computational load does not depend on the number of classes.

Performance of parallel execution is assessed by measuring the speedup value, computed as the ratio of the execution time experienced on a single node and the execution time on multiple nodes. All of the following figures show the average speedup computed on 20 runs, as well the 95% confidence intervals $\overline{X} \pm \lambda\overline{X}$, obtained with the student's t-distribution, where $\overline{X}$ is the mean speedup value of the runs. The value of λ was always lower than 0.05, which is a good indication of the reliability of the results. Figure 17 reports the speedup on three parallel nodes versus the number of items when varying the visibility radius. The algorithm scales very well: as the number of items increases up to 600,000, the speedup value increases up to values between 2.65 (with $R_V = 3$) and 2.9 (with $R_V = 10$).

We then analyzed the speedup when parallelizing the execution on up to nine nodes. Figure 18 reports the values of speedup versus the number of computing nodes and for different problem sizes when setting the visibility radius to 10. The good scalability and performance of the approach are confirmed, since (1) the speedup increases with the number of nodes, and (2) for a given number of nodes, the speedup increases with the problem size.

[4]Strong scaling investigates, for a fixed problem size, how the time to solution varies with the number of processors. Weak scaling, on the other hand, studies how the time to solution varies with processor count with a fixed problem size per processor.

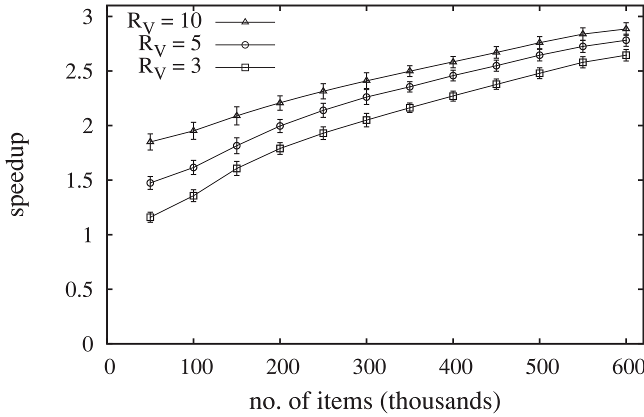


Fig. 17. Speedup on three nodes versus the number of items for different values of R_V. For all reported values, 95% confidence intervals are shown.

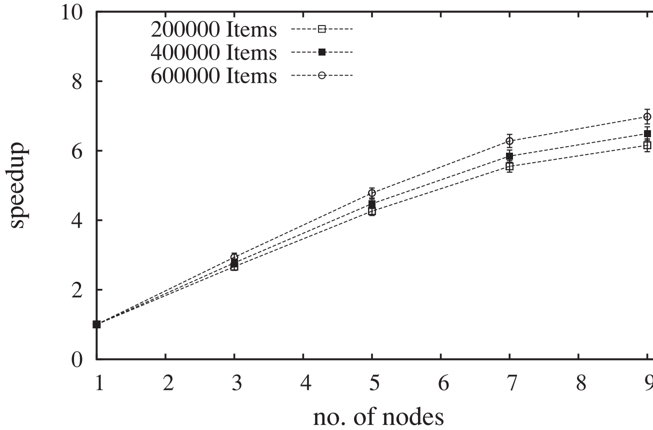


Fig. 18. Speedup versus the number of computing nodes for different problem sizes. For all reported values, 95% confidence intervals are shown.

In Figures 19 and 20, we show the effect of the pattern on the speedup: the former figure reports the speedup for parallel execution on three nodes versus the number of items, whereas the latter shows the speedup versus the number of nodes when setting the number of items to 600,000. In both cases, the visibility radius R_V is set to 10. These figures confirm that a pattern with a lower number of labels exhibits a better speedup value, as it reduces the synchronization cost (see Section 4.2). As a consequence, the Pattern P_C is the most effective in this scenario.

7. RELATED WORK

This article presents a new methodology that allows a wide class of algorithms to be ported transparently onto a parallel environment, particularly swarm algorithms in which individuals are represented by agents that operate in a territory. This section summarizes the state of the art regarding the parallelization of swarm algorithms, with particular focus on the techniques adopted for space partitioning and for the management of shared resources.

Ant-based clustering and sorting [Deneubourg et al. 1990b; Bonabeau et al. 1999], which is used in this article as an illustrative example for our methodology, is inspired

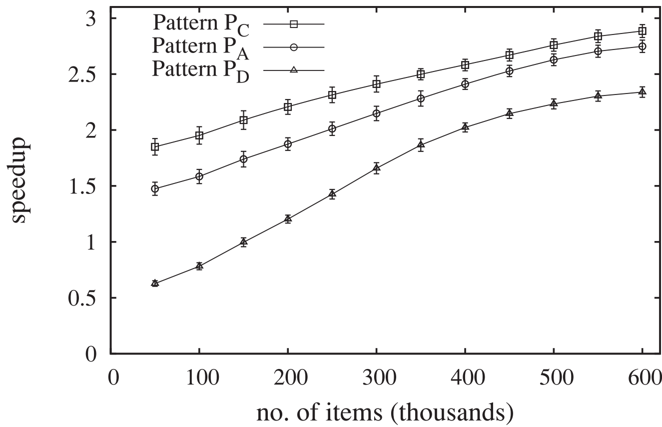


Fig. 19. Speedup on three nodes versus the number of items for different labeling patterns. For all reported values, 95% confidence intervals are shown.

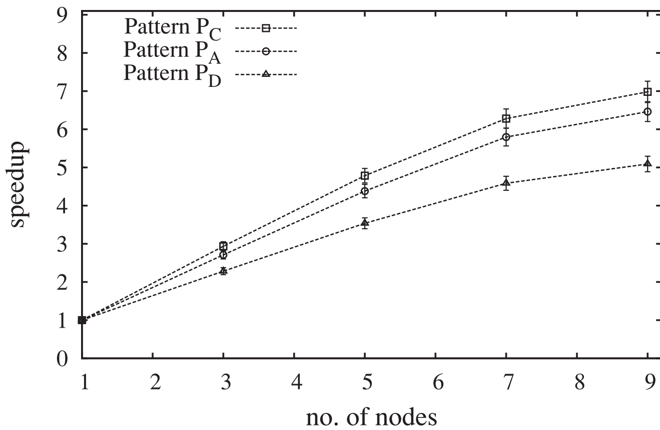


Fig. 20. Speedup versus the number of nodes for different labeling patterns. For all reported values, 95% confidence intervals are shown.

by the clustering of corpses and larval sorting observed in real ant colonies, and it is one of the best known example of swarm intelligence. The principles exploited by the basic versions of these algorithms, as well as successive versions devised along the years, such as ATTA [Handl et al. 2006] and So-Grid [Forestiero et al. 2008b], have been the subject of a large amount of studies. Some of these studies focused on the analysis of the performance of the algorithms [Handl et al. 2003] and on their ability to produce high-quality solutions [Handl et al. 2006]. Others focused on practical applications, such as data mining, graph partitioning, and reorganization/discovery of resources and documents in distributed information systems [Handl and Meyer 2002; Forestiero et al. 2008a].

Several strategies have been adopted to parallelize algorithms for ant-based clustering and sorting. In Yang et al. [2012], these algorithms are parallelized using the MapReduce programming model: the objects are partitioned into several parallel nodes, and partial results are collected by a central controller node. However, parallel nodes are isolated, and mobile agents (ants) are not allowed to migrate from one node to another, which limits their movements and hinders the faithful implementation of the

ant paradigm. In Albuquerque and Dupuis [2002], parallelization of ant operations is achieved through the use of cellular automata, and the space is partitioned into several regions, one per processor. The management of conflicts is addressed through arbitrary decisions that favor the ants on the basis of their positions.

ACO is a population-based metaheuristic that can be used to find approximate solutions to optimization problems [Dorigo and Stützle 2004; Dorigo and Di Caro 1999]. ACO has been applied to many application domains where there is the need to solve a problem that can be reduced to the well-known TSP, in which a salesman, starting from his hometown, wants to find the shortest tour that takes him through a given set of customer cities and then back home, visiting each customer city exactly once. Several artificial ants incrementally build solutions by moving on the graph, driven by pheromone traces. A systematic survey of the state of the art on parallel ACO implementations is offered in Pedemonte et al. [2011], where a taxonomy for classifying parallel ACO algorithms is proposed. In Cecilia et al. [2013] and Dawson and Stewart [2013], solutions of the TSP are obtained by using a parallel implementation of the ACO algorithm on GPU nodes. In these works, both the tour construction and pheromone update stages of the algorithm are achieved by using a data parallel approach. In Lin et al. [2007], ACO is parallelized by decomposing the problem in subcomponents, with each subgraph assigned to a different node. Manfrin et al. [2006a, 2006b] study the impact of communication when parallelizing a high-performing ACO algorithm for the TSP using message passing libraries. In particular, they analyze how different interconnection topologies affect the overall performance when the objective is to increase the quality of the solutions, given a fixed runtime. The adopted communication strategy involves the exchange of best-so-far solutions, which means the transmission of much less data than the exchange of information related to pheromone traces. In Craus and Rudeanu [2004], a framework for running sequential algorithms in a parallel environment is presented. An ACO algorithm is implemented on the framework to assess its performance in terms of speedup and communication cost. A configurable distributed architecture was proposed in Ilie and Badica [2013a] to provide an intuitive and simple mapping of ACO algorithms in a distributed environment. In this approach, the physical environment of ants is represented and implemented as a distributed multiagent system, and the movements of ants are modeled through messages that are exchanged asynchronously among the agents.

A general approach for the parallelization of swarm algorithms is described in Rouhipoura et al. [2010], where a GPU-based implementation of the bioinspired computing approach known as systemic computation (SC) is proposed. SC is a specific model of computation purposely designed to exploit many natural properties observed in biological systems including parallelism [Navlakha and Bar-Joseph 2015]. Another general approach is presented in Ilie and Badica [2013b], where the purpose is to distribute swarm intelligence algorithms that solve graph search problems on a computer network. This work proposes a novel distributed framework, for a class of swarm intelligence algorithms, which better exploits the inherently distributed nature of these algorithms.

A large number of recent papers focus explicitly on the problem of territory representation and handling. For example, in the field of swarm robotics [Brambilla et al. 2013], several works have employed spatial partitioning to achieve a wide set of goals. Dantu et al. [2011] and Mottola et al. [2014] use spatial partitioning to coordinate and parallelize distributed sensing in a swarm of mobile robots. The dispersion of the robots throughout the target space is tackled by providing a space abstraction and dividing the target area into regions. As in our approach, robot behaviors are assumed to be written in a location-agnostic manner so that they can be applied to any region in the target space. Pini et al. [2013] use spatial task partitioning to improve the performance

of a set of robots in a foraging scenario. The partitioning algorithm uses border areas to coordinate the robots: object transportation is performed by different robots, and each robot transports the object for a limited distance and hands it over to another robot, which continues transportation. This work uses ARGoS [Pinciroli et al. 2012], a robot simulator that can simulate thousands of robots in real time and is highly customizable. ARGoS divides the physical space in several regions to improve the performance of the simulation. Neither the robot code nor any modules of the simulator are affected by parallelization. ARGoS designers recognize that the use of mutexes or semaphores to manage conflicts and avoid race conditions can entail significant performance costs. The solution devised in ARGoS is to design the main loop as the composition of three phases (sense and control, act, physics) and to ensure, also through an appropriate space partitioning, that at each phase the robot components cannot be in conflict with each other.

The management of a shared territory is also addressed by many works in the field of situated multiagent systems—that is, systems in which the behavior of agents is strongly influenced by their positions in the territory and by their interactions with the surrounding environment [Bandini et al. 2002]. The management of the shared state representing the territory can become a bottleneck, limiting the overall performance when agent systems are executed in a parallel/distributed scenario [Logan 2007; Pawlaszczyk and Strassburger 2009]. The concept of spheres of influence is introduced in Logan and Theodoropoulos [2001] to manage a shared state in a distributed scenario. The purpose is to favor locality by putting information close to the agent that uses the information during execution. Spheres of influence are dynamically determined on the basis of the mutual interactions among agents and information. In the approach presented in Lees et al. [2005], shared data is maintained in a tuple space. The tuple space is partitioned by following a hierarchical schema based on the spheres of influence so as to avoid any bottleneck in managing the shared data. In Weyns and Holvoet [2004], the concept of synchronization regions is introduced to resolve conflicts among concurrent actions and reduce synchronization cost in a distributed setting. A *region* is a group of agents that act simultaneously and independently of other agents. Regions are determined by a decentralized synchronization algorithm that is executed when actions are performed.

All of the mentioned works that focus on the territory management show that the access to shared resources is one of the main issues that need to be addressed. To the best of our knowledge, the solutions proposed in the literature need an explicit effort on the part of the developer either to avoid the conflicts through mechanisms and protocols that exploit the characteristics of the specific domains (e.g., the presence of physical boundaries in swarm robotics) or for the definition of general explicit high-level synchronization mechanisms and primitives used to manage territory and shared data. The novelty of our approach resides in the general methodology presented, based on an original use of logical time, which ensures good performance while relieving the developer from taking care of the issues related to the management of shared state, territory handling, and performance optimization.

8. CONCLUSION AND FUTURE WORK

We presented and evaluated an approach that makes original use of the logical time concept to automate the parallelization of a wide class of swarm algorithms, those based on operations of mobile agents embedded in a territory. The article describes the mechanisms and policies used to partition the territory among parallel nodes and to manage shared data avoiding conflicts and data inconsistency. This relieves the developer of the burden of explicitly defining and managing high-level synchronization primitives (e.g., locks). The approach prevents potential conflict situations by assigning logical times

(labels) to agents and defining a conflict-free partial order of execution. The work offers a complete description of the software architecture, gives details about the different strategies used for the assignment of labels, and explains how such strategies can reduce the synchronization burden and ensure a fair algorithm evolution. Performance evaluation has been performed for a popular case, the ant-based spatial clustering and sorting of items, showing that our methodology, while preventing data consistency issues, ensures good scalability properties and can be adopted to speed up the algorithm execution when the problem is complex and/or its size is large.

Ongoing work aims at evaluating the pros and cons of different strategies for partitioning the territory and for assigning regions to computing nodes, such as monodimensional slicing as in this article versus bidimensional partitioning. Preliminary results show that the best choice depends on the size of the border areas and the behavior of the agents. Techniques for balancing the computational load of different regions are also under examination.

REFERENCES

P. Albuquerque and A. Dupuis. 2002. A parallel cellular ant colony algorithm for clustering and sorting. In *Cellular Automata*. Lecture Notes in Computer Science, Vol. 2493. Springer, 220–230.

S. Bandini, S. Manzoni, and C. Simone. 2002. Dealing with space in multi agent systems: A model for situated MAS. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*. 1183–1190.

G. Beni. 2005. From swarm intelligence to swarm robotics. In *Swarm Robotics*. Lecture Notes in Computer Science, Vol. 3342. Springer, 1–9.

E. Bonabeau, M. Dorigo, and G. Theraulaz. 1999. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York, NY.

E. Bonabeau, M. Dorigo, and G. Theraulaz. 2000. Inspiration for optimization from social insect behaviour. *Nature* 406, 6791, 39–42.

M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo. 2013. Swarm robotics: A review from the swarm engineering perspective. *Swarm Intelligence* 7, 1, 1–41.

D. Bueso. 2015. Scalability techniques for practical synchronization primitives. *Communications of the ACM* 58, 1, 66–74.

J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón. 2013. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing* 73, 1, 42–51.

F. Cicirelli, A. Forestiero, A. Giordano, and C. Mastroianni. 2014. An approach for scalable parallel execution of ant algorithms. In *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS'14)*. 170–177.

F. Cicirelli, A. Furfaro, A. Giordano, and L. Nigro. 2007. An agent infrastructure for distributed simulations over HLA and a case study using unmanned aerial vehicles. In *Proceedings of the 40th Annual Simulation Symposium (ANSS'07)*. 231–238.

F. Cicirelli, A. Giordano, and L. Nigro. 2015. Efficient environment management for distributed simulation of large-scale situated multi-agent systems. *Concurrency and Computation: Practice and Experience* 27, 3, 610–632.

M. Craus and L. Rudeanu. 2004. Parallel framework for ant-like algorithms. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing (ISPDC'04)*. 36–41.

K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh. 2011. Programming micro-aerial vehicle swarms with karma. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys'11)*. 121–134.

L. Dawson and I. Stewart. 2013. Improving ant colony optimization performance on the GPU using Cuda. In *Proceedings of the 2013 IEEE Congress on Evolutionary Computation (CEC'13)*. 1901–1908.

J.-L. Deneubourg, S. Aron, S. Goss, and J. M. Pasteels. 1990a. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior* 3, 2, 159–168.

J.-L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chrétien. 1990b. The dynamics of collective sorting robot-like ants and ant-like robots. In *Proceedings of the 1st International Conference on Simulation of Adaptive Behavior on From Animals to Animats*. 356–363.

M. Dorigo and G. Di Caro. 1999. The ant colony optimization meta-heuristic. In *New Ideas in Optimization*, D. Corne, M. Dorigo, F. Glover, D. Dasgupta, P. Moscato, R. Poli, and K. V. Price (Eds.). McGraw-Hill, UK, 11–32.

M. Dorigo, D. Floreano, L. M. Gambardella, F. Mondada, S. Nolfi, T. Baaboura, M. Birattari, et al. 2013. Swarmanoid: A novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics & Automation Magazine* 20, 4, 60–71.

M. Dorigo and T. Stützle. 2004. *Ant Colony Optimization*. MIT Press, Cambridge, MA.

J. Ferber. 1999. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison Wesley Longman.

A. Forestiero, C. Mastroianni, and G. Spezzano. 2008a. QoS-based dissemination of content in grids. *Future Generation Computer Systems* 24, 3, 235–244.

A. Forestiero, C. Mastroianni, and G. Spezzano. 2008b. So-Grid: A self-organizing grid featuring bio-inspired algorithms. *ACM Transactions on Autonomous and Adaptive Systems* 3, 2, Article No. 5.

P. Grassé. 1959. La reconstruction du nid et les coordinations inter-individuelles chez Belicositermes natalensis et Cubitermes sp. la theorie de la stigmergie: Essai d'interpratation du comportement des termites constructeurs. *Insectes Sociaux* 66, 41–84.

J. Handl, J. Knowles, and M. Dorigo. 2003. On the performance of ant-based clustering. In *Design and Application of Hybrid Intelligent Systems*, A. Abraham, M. Köppen, and K. Franke (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 104. IOS Press, 204–213.

J. Handl, J. Knowles, and M. Dorigo. 2006. Ant-based clustering and topographic mapping. *Artificial Life* 12, 1, 35–61.

J. Handl and B. Meyer. 2002. Improved ant-based clustering and sorting in a document retrieval interface. In *Parallel Problem Solving from Nature PPSN VII*. Lecture Notes in Computer Science, Vol. 2439. Springer, 913–923.

S. Ilie and C. Badica. 2013a. Multi-agent approach to distributed ant colony optimization. *Science of Computer Programming* 78, 6, 762–774.

S. Ilie and C. Badica. 2013b. Multi-agent distributed framework for swarm intelligence. *Procedia Computer Science* 18, 611–620.

D. Karaboga and B. Akay. 2009. A comparative study of artificial bee colony algorithm. *Applied Mathematics and Computation* 214, 1, 108–132.

J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks*, Vol. 4. 1942–1948.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. 2001. ASPECTJ. *Communications of the ACM* 44, 10, 59–65.

L. Lamport. 1978. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM* 21, 7, 558–565.

M. Lees, B. Logan, R. Minson, T. Oguara, and G. Theodoropoulos. 2005. Modelling environments for distributed simulation. In *Environments for Multi-Agent Systems*. Lecture Notes in Computer Science, Vol. 3374. Springer, 150–167.

Y. Lin, H. Cai, J. Xiao, and J. Zhang. 2007. Pseudo parallel ant colony optimization for continuous functions. In *Proceedings of the 3rd International Conference on Natural Computation (ICNC'07)*, Vol. 4. 494–500.

B. Logan. 2007. Evaluating agent architectures using simulation. In *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI'07)*. 40–43.

B. Logan and G. Theodoropoulos. 2001. The distributed simulation of multiagent systems. *Proceedings of the IEEE* 89, 2, 174–185.

M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo. 2006a. Parallel ant colony optimization for the traveling salesman problem. In *Ant Colony Optimization and Swarm Intelligence*. Lecture Notes in Computer Science, Vol. 4150. Springer, 224–234.

M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo. 2006b. Parallel multicolony ACO algorithm with exchange of solutions. In *Proceedings of the 18th Belgium–Netherlands Conference on Artificial Intelligence (BNAIC'06)*. 409–410.

L. Mottola, M. Moretta, K. Whitehouse, and C. Ghezzi. 2014. Team-level programming of drone sensor networks. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys'14)*. ACM, New York, NY, 177–190.

S. Navlakha and Z. Bar-Joseph. 2015. Distributed information processing in biological and computational systems. *Communications of the ACM* 58, 1, 94–102.

D. Pawlaszczyk and S. Strassburger. 2009. Scalability in distributed simulations of agent-based models. In *Proceedings of Winter Simulation Conference (WSC'09)*. 1189–1200.

M. Pedemonte, S. Nesmachnow, and H. Cancela. 2011. A survey on parallel ant colony optimization. *Applied Soft Computing* 11, 8, 5181–5197.

C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, et al. 2012. ARGoS: A modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence* 6, 4, 271–295.

G. Pini, A. Brutschy, C. Pinciroli, M. Dorigo, and M. Birattari. 2013. Autonomous task partitioning in robot foraging: An approach based on cost estimation. *Adaptive Behavior* 21, 2, 118–136.

C. W. Reynolds. 1987. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics* 21, 4, 25–34.

M. Rouhipoura, P. J. Bentleyb, and H. Shayanib. 2010. Fast bio-inspired computation using a GPU-based systemic computer. *Parallel Computing* 36, 10–11, 591–617.

K. Sycara. 1998. Multiagent systems. *Artificial Intelligence Magazine* 10, 2, 79–93.

C. Twomey, T. Stützle, M. Dorigo, M. Manfrin, and M. Birattari. 2010. An analysis of communication policies for homogeneous multi-colony ACO algorithms. *Information Sciences* 180, 12, 2390–2404.

D. Weyns and T. Holvoet. 2004. A formal model for situated multi-agent systems. *Fundamenta Informaticae* 63, 2–3, 125–158.

M. Wooldridge. 2002. *An Introduction to Multi-Agent Systems*. John Wiley & Sons.

Y. Yang, X. Ni, H. Wang, and Y. Zhao. 2012. Parallel implementation of ant-based clustering algorithm based on Hadoop. In *Proceedings of the 3rd International Conference on Advances in Swarm Intelligence (ICSI'12), Part I*. 190–197.

Received February 2015; revised February 2016; accepted February 2016