2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Strategies for Parallelizing Swarm Intelligence Algorithms

Franco Cicirelli
DIMES - University of Calabria
Rende (CS), Italy
email: f.cicirelli@dimes.unical.it

Gianluigi Folino, Agostino Forestiero, Andrea Giordano,
Carlo Mastroianni, Giandomenico Spezzano
ICAR-CNR, Rende (CS), Italy
email: {folino,forestiero,giordano,mastroianni,spezzano}@icar.cnr.it

***Abstract*—Swarm intelligence algorithms, based on multi-agent systems, are often used to solve complex problems that are not affordable through classical centralized/deterministic solutions. In many cases, to enhance the performance of such algorithms, the computation can be distributed to parallel/distributed nodes, in accordance with different strategies. Specifically, parallelization can be achieved either by partitioning the space in which agents operate among the nodes, or by assigning the entire space to each node but distributing input data through a sampling approach. Another choice is whether or not the management of conflicts is needed to prevent possible loss of data consistency. This paper discusses such issues, while referring to two well-known types of swarm intelligence algorithms – ants and flocking – and compares the mentioned strategies, evaluating the performance results in terms of speedup.**

Keywords*—*bio-inspired algorithms; multi-agent systems; parallel algorithms; swarm intelligence;

I. Introduction

Bio-inspired algorithms are widely exploited to solve a number of complex problems (combinatorial algorithms, task allocation, routing problems, graph partitioning, etc.) [2] and have been also adopted to provide advanced services in P2P networks [1], Grid systems and Cloud infrastructures. Most biological systems are founded on the *swarm intelligence* paradigm. A number of small and autonomous entities perform very simple operations driven by local information: for example, while searching for food an ant follows a pheromone substance deposited by another ant that has already discovered a food source; a bird adjusts its speed and direction by following the movements of nearby birds. From the combination of such operations a complex and intelligent behavior emerges: ants are able to establish the shortest path towards a food source; birds travel in large flocks and rapidly adapt their movements to the changing characteristics of the environment, etc. [2] [8].

Swarm biological algorithms can be executed by using situated multi-agent systems [22] [21] [11]: the behavior of insects and birds can be reproduced by agents that are situated in a hosting environment (territory) and perform simple operations. Agent-based systems may inherit useful and beneficial properties from biological counterparts, namely: (i) *self-organization*, since decisions are based on local information, i.e., without any central coordinator; (ii) *adaptivity*, since agents can react flexibly to the ever-changing environment; (iii) *stigmergy awareness* [14], since agents are able to interact and cooperate through the modifications of the environment that are induced by their operations.

Parallel/distributed execution of bio-inspired algorithms is often required to cope with the high demand of computational resources needed when the problem becomes more complex or its size increases. In a parallel/distributed scenario, the territory represents a huge shared variable of a concurrent system that needs a careful handling. Territory management requires conflicts and data consistency issues to be addressed. Moreover, frequent access to territory information may easily become a bottleneck impairing the overall system performance and scalability.

This work describes and compares two different methodologies for parallelizing SI algorithms, which can be used depending on the specific requirements and features of the environment and of the problem to be solved. The first methodology splits the entire space in which the agents move into regions. Each region, along with the data and the agents residing on it, is allocated for execution on a distinct computing node. The borders of the regions require to be kept aligned and updated. This methodology is further refined in two variants: *space partitioning without conflict avoidance* and *space partitioning with conflict avoidance*. The former can be used when the algorithm to execute is not affected by concurrency issues. The latter, instead, is used to transparently avoid that concurrent agents allocated on different computing nodes may undergo conflicting actions. This methodology partitions the data among different nodes, while the space is not partitioned, but it is shared/replicated over all the nodes. In order to better illustrate the advantages and drawbacks of the two approaches, two examples of SI algorithms are described: a flocking algorithm for searching objects and an ant algorithm for spatial clustering. Experiments show that both methodologies are efficient and scale well, and help to identify which of the two is the most suitable for any specific problem.

The paper is structured as follows: Section II and III illustrate, respectively, the flocking and the ant algorithm. In Section IV, the two different methodologies used for parallelizing SI algorithms are described and compared by considering their effectiveness when applied to the flocking and the ant algorithms. Section V shows the performance in terms of speedup of the different strategies. Section VI concludes the paper and gives hints on future research directions.

II. A flocking algorithm for searching spatial data

In this section, a multi-agent stochastic search algorithm is presented, which has the advantage of being easily imple-

1066-6192/15 $31.00 © 2015 IEEE
DOI 10.1109/PDP.2015.101

mentable on parallel and distributed machines and is robust with respect to the failure of individual agents. First, the rules governing the basic flock model [20] are explained; then, the algorithm is specialized to the problem of searching objects in a 2-dimensional space.

The flocking algorithm was originally proposed by Reynolds [20] as a method for simulating the flocking behavior of birds on a computer both for animation and as a way to study emergent behavior. Flocking is an example of emergent collective behavior: there is no leader nor global control. Each agent/bird reacts only to flock mates within a certain small radius, and the intelligent behavior emerges from such local interactions. The basic flocking model consists of three kind of simple steering behavior: (i) **Separation** gives an agent the ability to maintain a certain distance from its neighbors, which prevents agents from crowding too closely together; (ii) **Cohesion** supplies an agent with the ability to cohere (approach and form a group) with other nearby agents. Agents are attracted towards the center od their group through a steering force that is applied in the direction of that average position; (iii) **Alignment** gives an agent the ability to align with other nearby characters. Steering for alignment can be computed by finding all agents in the local neighborhood and averaging together the 'heading' vectors of the nearby agents.

Different variants of the basic flock algorithms can be used to cope with the problem of searching interesting objects in spatial data. An adaptive algorithm able to discover these points in parallel was introduced in [12]. This algorithm uses a modified version of the standard flocking algorithm, which incorporates the capacity for learning that is present in many insect colonies, such as ants or bees. In the modified algorithm, the agents are transformed into hunters with a foraging behavior that allows them to explore spatial data efficiently.

The algorithm starts with a fixed number of agents that occupy a randomly generated position in a 2-dimensional space. Each agent moves around the spatial data, testing the neighborhood of each location in order to verify whether a point can have some desired properties. Each agent follows the rules of movement described in Reynolds' model. In addition, the model considers four different kinds of agents, classified on the basis of some properties of data in their neighborhood. Different agents are characterized by a different color: *red*, revealing interesting patterns in the data, *green*, a medium one, *yellow*, a low one, and *white*, indicating a total absence of patterns.

The main idea behind this approach is to take advantage of the colored agents in order to explore more accurately the most interesting regions (signaled by the red agents) and avoid the ones without interesting points (detected by the white agents). Red and white agents stop moving in order to signal this type of region to the others, while green and yellow ones fly to find denser zones. Indeed, each flying agent computes its heading by taking the weighted average of alignment, separation and cohesion (as illustrated in Figure 1).

The following are the main features which make this model different from Reynolds' model:

- *Alignment* and *cohesion* do not consider yellow agents, since they move in a not very attractive zone.

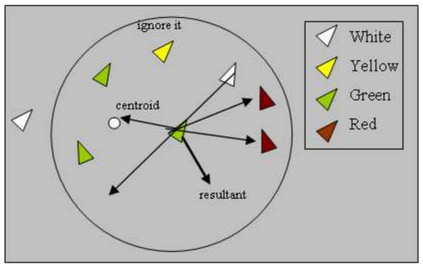


Fig. 1. Computing the direction of a green agent.

- *Cohesion* is the resultant of heading towards the average position of the green flockmates (centroid), of the attraction towards reds, and of the repulsion from whites, as illustrated in Figure 1.
- A *separation* distance is maintained from all the agents, apart from their color.

In Figure 2, we summarized the pseudo code of the overall algorithm with the main functions better explained in the table I.

```
for i=1 . . . MaxIterations
    foreach agent (yellow, green)
        age=age+1;
        if (age > Max_Life)
            generate_new_agent();die();
        endif
        if (not visited (current_point))
            property = compute_local_property(current_point);
            mycolor= color_agent(property);
        endif
    end foreach
    foreach agent (yellow, green)
        dir= compute_dir();
    end foreach
    foreach agent (all)
        switch (mycolor){
            case yellow, green: move(dir, speed(mycolor)); break;
            case white: stop(); generate_new_agent(); break;
            case red: stop(); generate_new_close_agent(); break; }
    end foreach
end for
```

Fig. 2. The pseudo-code of the adaptive flocking algorithm.

Function	Meaning
generate_new_close_agent()	generate a new agent in a random position (close to the current)
generate_new_agent()	generate a new agent in a random position of the search space
stop()	stop the agent
die()	kill the agent
compute_local_property(P)	compute the desired property at the point P
color_agent(prop)	color the agent on the basis of the property $prop$
speed(col)	compute the speed of the agent depending on the color col
compute_dir()	compute the new direction of the agent using the modified Reynolds' model
move(dir, s)	move the agent with direction dir and speed s

TABLE I. MEANING OF THE MAIN FUNCTIONS USED IN THE PSEUDOCODE.

The color of the agent is assigned on the basis of the desired property at the point in which it falls; the assignment is made on a scale going from white ($property = 0$) to red ($property > threshold$), passing for yellow and green, corresponding to intermediate values. Yellow and green agents compute their direction, according to the rules previously described, and move following this direction. Green agents move more slowly than yellow agents since the former explore more interesting regions. A formal description of this flocking algorithm and its application to the problem of clustering large dataset is presented in [12].

The described flock algorithm is inherently parallel and is apt to be implemented on parallel/distributed architectures, as will be detailed in Section IV.

III. An ant algorithm for clustering spatial data

The objective of the basic ant algorithm presented in [2] is to cluster items in a two-dimensional space. The space is partitioned in cells, forming a two-dimensional grid. Each ant moves hopping between adjacent cells and has visibility over the items deposited in its visibility area.

In the simplest scenario, in which items are identical and are spread, the goal is to create regions in which items are accumulated, leaving empty regions in between. This basic version can be specialized in many ways depending on the application. In a very common and significant variation, items belong to a number of predefined different classes, and the objective becomes to spatially sort items, i.e., separate items of different classes and clustering items of the same class. This problem has a large number of applications in several domains, from the organization of physical objects performed by robots to data analysis and clustering in distributed information systems.

In the following, we briefly describe the approach presented in [13]. Each ant contributes to the reorganization by *picking* and *dropping* items from/to the cells. The ants perform their operations following time-stepped advancements. At each time-step, every ant performs a single operation, either a hop towards an adjacent cell or a drop/pick attempt. The *pick* and *drop* operations are driven by corresponding probability functions, which are inspired by the mechanisms introduced in [9], and later elaborated and discussed in [2] and [18], to emulate the behavior of some species of ants that cluster and sort items in their environment.

The probability of picking an item of a given class from a cell must decrease as items of the same class are accumulated in the visibility area centered in that cell. This ensures that as soon as the equilibrium condition is broken (i.e., items belonging to different classes begin to be accumulated in different areas), a further reorganization of items is increasingly fostered. The P_{pick} probability function, defined in formula (1) below, and inspired by the pick probability defined in [2], aims to achieve the spatial separation of items belonging to different classes.

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

For each class c of items, the fraction f_c is computed as the number of items of class c, accumulated in the cells within the visibility area, divided by the overall number of items of *all* classes that are accumulated in the same area. As the local area accumulates more items of a class, with respect to other classes, f_c increases and the value of the pick probability for this class becomes lower, and vice versa. This has to effect of inducing agents to pick items that are uncommon in the local area, and leave items of the class that is being accumulated. The parameter k_p is assigned a non-negative value and is used to tune the clustering effort. In the tests performed in this work it is set to 0.1, as in [2].

After picking an item, the agent travels the system hopping between adjacent cells, and at any new cell it must decide whether or not to drop the item. Like the pick function, the drop function is first used to break the initial equilibrium and then to strengthen the spatial clustering of items. The *drop* probability for a class, shown in formula (2) below, *increases* as the local area accumulates items of this class. In (2), the fraction f_c is defined as in formula (1), whereas the parameter k_d is set to 0.3 [2].

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

The effectiveness of the clustering algorithm is evaluated through a spatial entropy function, based on the well-known Shannon's 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 mapped in the area around l. In (3), f_c is the fraction of items of class c ($c = 1...C$, where C is the number of predefined classes) that are located in the visibility area with respect to the overall number of items located in the same area. The E_l function 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 area centered in l has accumulated a large number of items belonging to one specific class. As shown in formula (4), the overall entropy E is defined as the average of the entropy values E_l computed at all the system cells (the number of cells is equal to N_l).

$$E_l = \frac{\sum_{(c=1...C)} f_c \cdot \lg \frac{1}{f_c}}{\lg C} \tag{3}$$

$$E = \frac{\sum_l E_l}{N_l} \tag{4}$$

This kind of ant algorithm can be transparently executed in a parallel environment while ensuring scalability and preventing consistency issues, as discussed in Section IV.

IV. An agent-based framework for parallel SI algorithms

As the problem size increases, it may be convenient to parallelize or distribute the execution of an entire SI algorithm. In literature, there are different strategies for parallelizing swarm intelligent algorithms, mainly oriented towards particular types of algorithm, specially ant-based (see [10] [19] [15] [23]) and flock-based (see [7] [16]), but to the best of our knowledge there is no unified discussion on parallelizing swarm intelligent algorithms.

Actually, most of the SI algorithms can be suitably modeled and implemented using a multi-agent system, where agents interact in a spatial environment. In literature, this kind of agents are often called *situated agents*, i.e., an agent owns spatial coordinates and is embedded into the territory (spatial environment), where it lives and moves [22] [11]. Furthermore, the space can also be populated by some data, which is usually processed by agents in order to accomplish a task, i.e., clustering, sorting, etc..

Before going into the issues related to parallelization, we need to introduce the notions of visibility and action radius. The *visibility radius* (VR) is exploited in order to delimit the area within which an agent is able to perceive its surrounding space. The *action radius* (AR) defines the area that can be modified by a single operation of an agent located in a given position.

The way SI algorithms can be parallelized ultimately depends on the interaction patterns allowed among the agents, the space and the data. Indeed, in a parallel/distributed scenario, frequent concurrent accesses of agents to the territory can easily become a bottleneck that limits system performance and scalability. Another aspect to be considered is that data objects and agents can be either partitioned in accordance with agents' spatial positions (*space partitioning*) or they can be randomly sampled and partitioned over the nodes of the parallel/distributed architecture (*data partitioning*).

Based on these considerations, three strategies for parallelization are proposed in the following, two based on space partitioning (with or without conflict management) and one based on data partitioning.

A. Space partitioning

With the space partitioning strategy, the whole territory is split into *regions*. Each region, along with the data and the agents residing on it, is allocated for execution on a distinct computing node [4] [5] [6]. The *borders* of the regions, i.e., the portions of the space which are adjacent among contiguous regions, require to be kept aligned and updated. Partitioning the territory among multiple nodes is a key to avoid the access to shared content to become a bottleneck. Splitting the territory favors system scalability as the size of the problem increases (in terms of number of agents, data size or territory size) and more computing nodes are used to speed up the execution. The strategy based on partitioning can be further refined in *space partitioning without conflict avoidance* and *space partitioning with conflict avoidance*. The former can be used when the algorithm is not affected by concurrency issues. The latter, instead, is used to transparently avoid that concurrent agents allocated on different computing nodes may undergo conflicting actions.

1) Space partitioning without conflict avoidance: In order to use the partitioning schema without conflict avoidance, the basic assumption is that the algorithm be *conflict-free* by itself. Basically, this property is related to the state of the spatial environment and the interaction between agents and the environment. If agents can modify the state of the territory, issues related to the concurrent access should be taken into account thus preventing loss of consistency in the data. For example, let us consider an ant-based algorithm (as the one presented in Section III) in the space partitioning scenario. The agents carry objects from place to place across the territory so two concurrent agents (i.e., executing on different computing nodes) may pick up the same object at the same time or collapse moving towards the same cell. On the basis of these considerations, we can claim that this kind of algorithm needs conflict avoidance mechanisms such as those described in IV-A2. Conversely, a flock-based algorithm, as the one presented in Section II, can be parallelized without conflict avoidance. Indeed, flocks only need to read the spatial environment in order to set their flying activities. In other words, each flock can only modify its own state while the territory is just the algorithm input and does not change during the execution. In addition, flocks are already equipped with a specific conflict avoidance mechanism based on the *separation* distance as described in Section II.

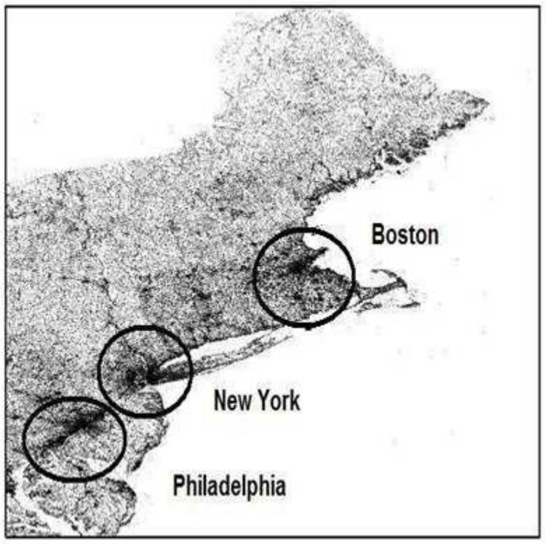


Fig. 3. The North-East dataset

For the sake of clearness, we take as an example the dataset North-East (see Figure 3), widely used for the task of clustering the three populated towns of Boston, New York and Philadelphia. Figure 4 graphically suggests how the territory may be partitioned in regions. Each agent is executed by the computing node hosting the region where the agent is located. Agent migration is required when an ant moves from a region to another. During exploration, an agent requires to retrieve information about its neighborhood, i.e., the portion of the territory covered by its visibility area. These interactions are *local* when the neighborhood is comprised in the same region and hence managed by the same computing node. Conversely, when the neighborhood involves the same portion of the territory belonging to other regions, *remote* interactions

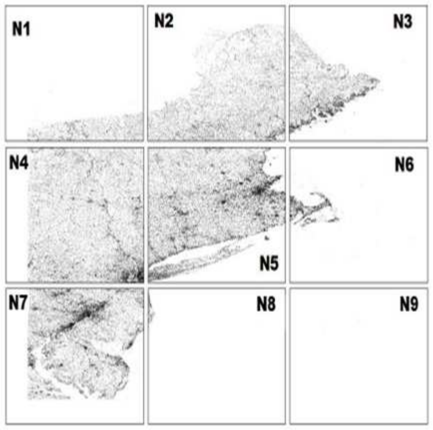


Fig. 4. Search space decomposed among 9 nodes.

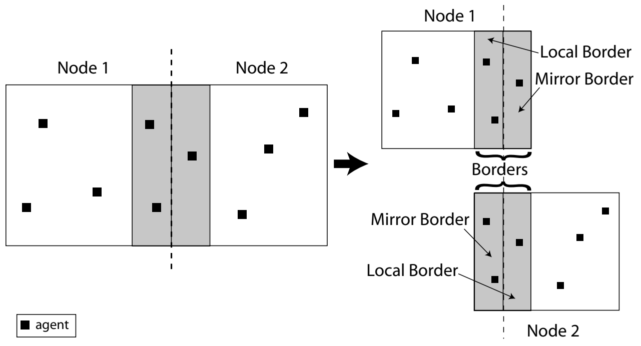
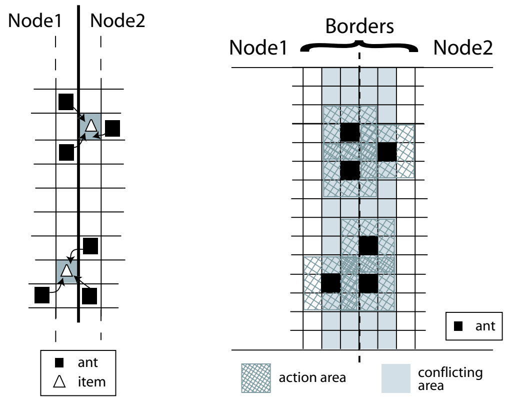


Fig. 5. Border areas of two adjacent nodes.

– with inter-node information exchange – would be required if this issue is not properly managed. In our approach, in order to avoid remote agent operations, the edge portion of a region is replicated in adjacent nodes. This edge portion is referred to as a *border* of the region. As shown in Figure 5, the border area of a given region (consider Node 1 in the figure) 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 sent to the mirror border of the adjacent node, i.e., Node 2. Analogously, the mirror border of Node 1 includes information replicated from the local border of Node 2. The width of borders is determined by the visibility radius. Borders are kept aligned by exchanging update messages among the computing nodes that manage adjacent regions. This strategy allows all the sensing operations to be performed locally.

2) Space partitioning with conflict avoidance: In this scenario, issues relevant to *consistency* and *conflict resolution* on shared data are handled. Usually, conflict resolution and consistency are achieved by resorting to synchronization primitives (e.g. based on locks), which, though, may present two main drawbacks: (i) they may hinder the transparency of the parallelization procedure, since the developer is compelled to cope with the management of such primitives, and (ii) they may negatively impact on performance and scalability. Our approach allows the mentioned issues to be tackled by using a methodology, based on *logical time* [4] which is



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

Fig. 6. Scenario with ants conflicting on the borders of two nodes

able to transparently enforce a conflict-free and fair execution order on concurrent actions. The territory is modeled as a bi-dimensional discrete grid of cells. Update messages are still used to keep the borders aligned.

Figure 6(a) shows an example of conflicting scenario where some ants compete to pick up the same items contained in the grey cells. As the ants operate concurrently, two of them may try to pick up the same item: if both pick operations are actually performed, this will lead to an inconsistent state of the algorithm. Each node operates sequentially, i.e., a non-preemptive interleaved execution of agent actions is adopted. Therefore, agents execute concurrently only if they are located on different regions. As shown in Figure 6(b), two agents are *potentially conflicting* when they belong to different regions and their action radii overlap. The *conflicting area* is defined as the portion of the territory that can host potentially conflicting ants (see the grey part of Figure 6(b)).

To prevent conflicts we borrow the notion of *logical time* from the distributed system field. The logical time concept [17] is typically used to prevent causality-constraint violations in distributed systems. In our approach, instead, we exploit it as a tie-breaking mechanism that prevents conflicts [5]. The idea underlying our approach consists in establishing a partial order of agent executions during a given time step such that no potentially conflicting agents will execute concurrently. The problem reduces to assigning, at any given time step, labels (natural numbers) to agents such 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 time step, every computing node executes the agents located in the corresponding region, respecting the label ordering discussed here: it executes all the ants with label 1 (the order among them is inessential), then those with label 2, etc. To ensure the algorithm consistency, the nodes must synchronize among them with respect to time steps and ordering labels.

An easy fashion to perform a conflict-free labeling of agents is: first assign labels to the cells belonging to the conflicting area and then assign each agent the label of the cell where the ant is located. Figure 7 shows the schema adopted in this work.

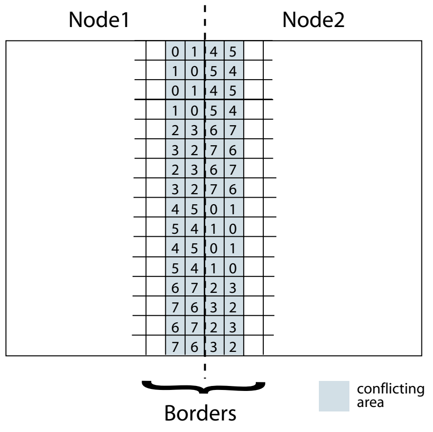


Fig. 7. Schema adopted for assigning labels to cells in the conflicting area

The choice of adopting the schema of Figure 7 derive, as detailed in [4] [5], from two requirements: (i) limit the number of labels, so as to reduce the number of synchronization points and (ii) assign the labels to cells so that two adjacent regions will always execute comparable numbers of operations at each label, i.e., at each logical time. The latter requirement helps to maximize the concurrency degree. The schema of Figure 7 is established by using a backtracking-based algorithm executed offline with respect to the execution of the SI algorithm so as not to affect system performance.

B. Data partitioning

The strategy consists of partitioning the data among different nodes, while the space is shared/replicated over the nodes. More in detail, each node handles the whole space, which contains only a subset of the data objects to be processed. Typically the data can be portioned equally among the nodes, but a different strategy could consider the assignment of a portion of data proportional to the computation power of the nodes.

The computation is synchronized only at the beginning, when data is sampled and distributed, and at the end, when results are collected. Depending on the particular needs of the algorithm, partial results or useful info could be exchanged after each iteration. Figure 8 illustrates an example of this approach for the North-East dataset.

C. Discussion

Using the data partitioning strategy, the load is well balanced, but other problems could raise. First of all, it is necessary to choose the right threshold (i.e., the right number of nodes on which divide the data), otherwise, by dividing data among the nodes, part of the data could be not efficiently processed. Furthermore, while by partitioning the space, the final results of the algorithm will be the same as the sequential case, this does not apply to this strategy. Indeed, the results could depend on the number of agents chosen to process the data, on the portion of data, which could not be sufficient to obtain determined results, and also on the number of nodes.

Conversely, using space partitioning, the load balancing constitutes a critical and well known issue [3]. In fact, for instance in the Figure 4, node 9 does not work at all and nodes 1, 6 and 8 work on a very limited portion of data. To obtain a better load balancing, choosing more complicated decomposition would help, but this would require a deeper knowledge of the domain of data and would bring to a more complex communication pattern among neighboring nodes, increasing the communication overhead.

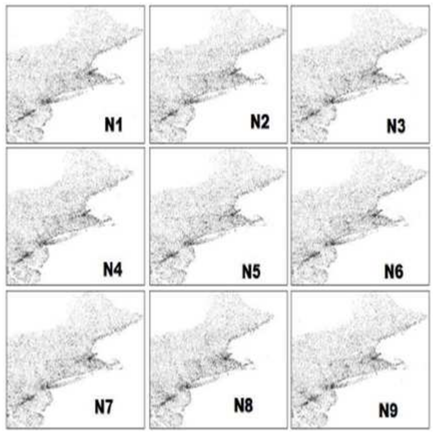


Fig. 8. Search space decomposed among 9 nodes (partitioning the data).

Anyway, in situations in which for reasons of privacy or because data are too large to be moved or because data are already logically and physically distributed, the space partitioning solution should be strongly taken into account. For example, this is the case of a scenario in which huge amounts of data are stored in autonomous, geographically distributed sources over networks with limited bandwidth and large number of computational resources.

V. Experimental Results

This section evaluates the achievable execution performance of the discussed strategies for parallelizing SI algorithms. For each kind of strategy, a different set of experiments was performed, and results are shown in the next subsections. Performance of parallel execution is assessed by measuring the speedup value, computed as the ratio of the execution time measured on a single node and the execution time on multiple nodes. The experiments were carried out on a cluster, in which each computing node has two CPU Intel(R) Xeon(R) CPU E5-2670 2.60GHz and 128GB RAM. The nodes are interconnected by a high performance Infiniband network.

A. Speedup for the case of space partitioning

This strategy has been evaluated for a typical clustering context where the ant algorithm of section III is used to sort items belonging to different classes. The items are spread in a two-dimensional grid of 120 x 100 cells, and an average number of items per cell equal to 50. The items belong to a number of classes C equal to 9 and are randomly spread over the cells in accordance to a uniform distribution. The number of ant-like agents is equal to 600,000. The visibility radius VR is set to 10. The value of the overall entropy, as defined

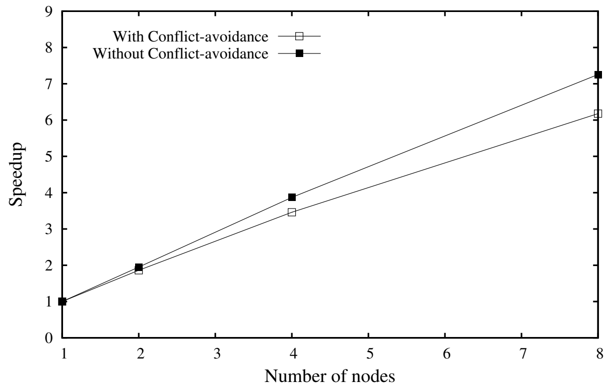


Fig. 9. Speedup vs. the number of computing nodes.

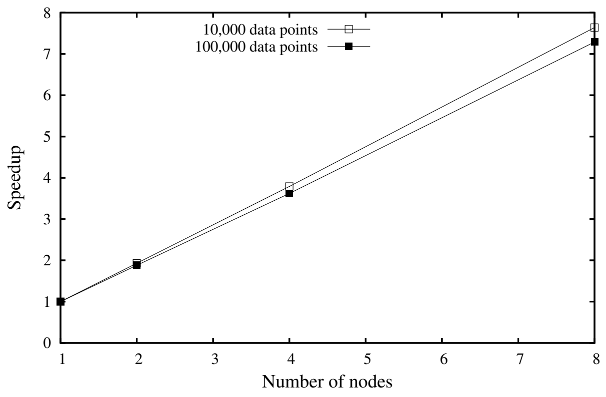


Fig. 10. Speedup vs. the number of computing nodes for 10,000 and 100,000 data points.

in expression (4), is used as stop criteria for the algorithm. The entropy is computed every 1000 time steps, and decreases as the algorithm proceeds, confirming its effectiveness. The system is considered stable, i.e., the algorithm terminates, when 10 consecutive values of the entropy differ among them no more than 1%. The scalability is evaluated with two sets of experiments: in the first set we used the strategy for conflict avoidance described in Section IV-A2, while in the second set we did not manage conflicts[1]. For both sets of experiments, we varied the number of parallel nodes up to 8.

Figure 9 reports the values of speedup vs. the number of computing nodes for the two sets of experiments. Experimental results show the good scalability and performance of the approach, and confirm that the management of conflicts introduces a slight overhead, which is witnessed by a lower speedup value.

B. Speedup for the case of data partitioning

This strategy does not present particular problems in terms of scalability, since in most cases the flocking algorithm does not require to exchange data among the nodes. However, depending on the specific problem that the flocking algorithm tries to solve, it could be necessary to exchange partial results or some data. In our experiments, we consider a simple case of a flock searching for points with a desired property, i.e., that the density of neighboring points within a given radius exceeds a predefined threshold. The points presenting this property are exchanged for each iteration of the flocking algorithm. Note that the algorithm constitutes the basis for the implementation of a clustering algorithm or an outlier detection algorithm.

Using the North-East dataset, two sets of experiments are conducted, considering respectively a total number of 10,000 and 100,000 points randomly sampled from the original dataset. All the data are partitioned over a number of nodes varying from 1 to 8. A visibility radius of 10 was considered.

Figure 10 shows the values of speedup for the two test suites. It is clear that the scalability is almost linear for both cases, and it is mainly due to the limited overhead of communication of the strategy. The case of 100,000 scales slightly worst because the number of points verifying the property and that have to be exchanged is larger – hence, the time needed for data transmission increases – while the time needed to compute the density is hardly affected by the number of points.

[1]Of course the possible presence of conflicts can affect the correct behavior of the application. Here, the goal of the tests is merely to assess performances.

VI. Conclusions

Two different strategies for parallelizing SI algorithms are presented, the first based on the partitioning of the space in which the agents move, and the second on the partitioning of the data to be processed by the agents. In addition, a methodology for avoiding concurrency issues is used to transparently avoid that concurrent agents allocated on different computing nodes may undergo conflicting actions. A flocking algorithm for searching objects and an ant algorithm for spatial clustering are used to better illustrate the advantages and drawbacks of the two approaches.

Experiments show that both methodologies are efficient and scale well. In particular, space partitioning is more convenient when there are privacy requirements, when moving data is expensive/unfeasible or when the management of conflicts is essential. Moreover, space partitioning guarantees that the obtained results are the same as in the sequential case, while data partitioning ensures better results in terms of scalability.

Acknowledgment

This work has been partially supported by MIUR-PON under project PON03PE_00032_2 within the framework of the Technological District on Cyber Security and project PON03PE_00050_2 within the framework of the Technological District on Smart Home (DOMUS).

References

[1] Ozalp Babaoglu, Hein Meling, and Alberto Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proc. of the 22 nd International Conference on Distributed Computing Systems ICDCS'02*, pages 15–22, Washington, DC, USA, 2002. IEEE Computer Society.

[2] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: from natural to artificial systems.* Oxford University Press, New York, NY, USA, 1999.

[3] Mario Cannataro, Salvatore Di Gregorio, Rocco Rongo, William Spataro, Giandomenico Spezzano, and Domenico Talia. A parallel cellular automata environment on multicomputers for computational science. *Parallel Computing*, 21(5):803–823, 1995.

[4] F. Cicirelli, A. Giordano, and L. Nigro. Distributed simulation of situated multi-agent systems. In *Proc. of the IEEE/ACM 15th International Symposium on Distributed Simulation and Real Time Applications*, pages 28–35, Washington, DC, USA, 2011.

[5] F. Cicirelli, A. Giordano, and L. Nigro. Efficient environment management for distributed simulation of large-scale situated multi-agent systems. *Concurrency and Computation: Practice and Experience*, 2014.

[6] Franco Cicirelli, Agostino Forestiero, Andrea Giordano, and Carlo Mastroianni. An approach for scalable parallel execution of ant algorithms. In *International Conference on High Performance Computing & Simulation, HPCS 2014*, Bologna, Italy, July 2014.

[7] Xiaohui Cui and Thomas E. Potok. A distributed agent implementation of multiple species flocking model for document partitioning clustering. In *Cooperative Information Agents X, 10th International Workshop, 2006, Edinburgh, UK, September 11-13, 2006, Proceedings*, pages 124–137, 2006.

[8] Prithviraj Dasgupta. Intelligent agent enabled peer-to-peer search using ant-based heuristics. In *Proc. of the International Conference on Artificial Intelligence IC-AI'04*, pages 351–357, 2004.

[9] J. L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chrétien. The dynamics of collective sorting robot-like ants and ant-like robots. In *Proc. of the First International Conference on Simulation of Adaptive Behavior on From Animals to Animats*, pages 356–363, Cambridge, MA, USA, 1990. MIT Press.

[10] Gianni Di Caro and Marco Dorigo. Antnet: Distributed stigmergetic control for communications networks. *J. Artif. Int. Res.*, 9(1):317–365, December 1998.

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

[12] Gianluigi Folino, Agostino Forestiero, and Giandomenico Spezzano. An adaptive flocking algorithm for performing approximate clustering. *Inf. Sci.*, 179(18):3059–3078, 2009.

[13] Agostino Forestiero, Carlo Mastroianni, and Giandomenico Spezzano. So-grid: A self-organizing grid featuring bio-inspired algorithms. *ACM Transactions on Autonomous and Adaptive Systems*, 3(2), May 2008.

[14] P. Grassé. La reconstruction du nid et les coordinations inter-individuelles chez bellicosi-termes natalensis et cubitermes sp. la theorie de la stigmergie: Essai d'interpretation des termites constructeurs. *Insectes Sociaux*, 6:41–84, 1959.

[15] Timur Keskinturk, Mehmet B. Yildirim, and Mehmet Barut. An ant colony optimization algorithm for load balancing in parallel machines with sequence-dependent setup times. *Computers & Operations Research*, 39(6):1225 – 1235, 2012.

[16] Marek Kisiel-Dorohinicki. Flock-based architecture for distributed evolutionary algorithms. In *Artificial Intelligence and Soft Computing - ICAISC 2004, 7th International Conference, Zakopane, Poland, June 7-11, 2004, Proceedings*, pages 841–846, 2004.

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

[18] Marc Martin, Bastien Chopard, and Paul Albuquerque. Formation of an ant cemetery: swarm intelligence or statistical accident? *Future Generation Computer Systems*, 18(7):951–959, 2002.

[19] Martn Pedemonte, Sergio Nesmachnow, and Hctor Cancela. A survey on parallel ant colony optimization. *Applied Soft Computing*, 11(8):5181 – 5197, 2011.

[20] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM Press.

[21] Katia Sycara. Multiagent systems. *Artificial Intelligence Magazine*, 10(2):79–93, 1998.

[22] M. Wooldridge. *An introduction to multi-agent systems*. John Wiley & Sons, Ltd., 2002.

[23] Yan Yang, Xianhua Ni, Hongjun Wang, and Yiteng Zhao. Parallel implementation of ant-based clustering algorithm based on hadoop. In *Proc. of the 3rd Int. Conference on Advances in Swarm Intelligence - Part I*, ICSI'12, pages 190–197, Shenzhen, China, October 2012. Springer-Verlag.