Distributed Load Balancing for Parallel Agent-based Simulations

Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara and Vittorio Scarano
ISISLab, Dip. di Informatica ed Applicazioni "Renato M. Capocelli"
Università degli Studi di Salerno, 84084, Fisciano (SA), Italy
{cosenza,cordasco,dechiara,vitsca}@dia.unisa.it

***Abstract*—We focus on agent-based simulations where a large number of agents move in the space, obeying to some simple rules. Since such kind of simulations are computational intensive, it is challenging, for such a contest, to let the number of agents to grow and to increase the quality of the simulation. A fascinating way to answer to this need is by exploiting parallel architectures.**

In this paper, we present a novel *distributed load balancing* schema for a parallel implementation of such simulations. The purpose of such schema is to achieve an high scalability. Our approach to load balancing is designed to be lightweight and totally distributed: the calculations for the balancing take place at each computational step, and influences the successive step.

To the best of our knowledge, our approach is the first distributed load balancing schema in this context.

We present both the design and the implementation that allowed us to perform a number of experiments, with up-to $1,000,000$ agents. Tests show that, in spite of the fact that the load balancing algorithm is local, the workload distribution is balanced while the communication overhead is negligible.

Keywords—Distributed Computing, Parallel Algorithms, Behavioral Simulations.

I. Introduction

The simulation of groups of agents moving in a virtual world is a topic that has been investigated since the 1980s. A widespread approach to this kind of simulations has been introduced in [16] and it takes inspiration from *particles system* [13]. In a particle system there is an *emitter* that generates a number of particles that move accordingly to a set of physics-inspired parameters (e.g. initial velocity, gravity). The particle system approach is expanded with the purpose of simulating a group of more complex entities, dubbed *autonomous agents*, whose movements are related to social interactions among group members.

A classical example of use of this approach is the flocking model proposed by Reynolds [16], which allows to simulate a flock of birds in the most natural possible way. Elements of this simulated flock are usually named *boids* (from *birdoid*) and got instilled a range of *behaviors* that induces some kind of *personality*. The behaviors are, in the most of cases, simply geometric calculations performed on each boid. Commonly, every boid is subject to three different behaviors: *separation* from other boids, *alignment* to other boids flight direction, and *cohesion* to other boids. Each of these behaviors is rendered by a force that is applied on the boid, and whose intensity depends on a fixed number of near flockmates, within a given radius which determines the boid's Area Of Interest (AOI). Actual implementations of the boid model may vary [6], [11], [15], [19] but the idea is the following: at every step of the simulation, for every boid b and for each behavior in the personality the system calculates a request to accelerate in a certain direction as the result of a weighted sum of all the forces applied on b.

As a counterpart of the realism achieved, the computation complexity of the model is $O(n^2)$, where n is the number of agents in the simulation. A way to achieve good performances, as the number of the agents increases, we can distribute the calculation on a number of workers. Several parallel implementation of the flocking model have been proposed (cf. [6], [7], [15], [24]). A good parallel implementation should strive to achieve two conflicting goals: (1) balance the overall load distribution, and (2) minimize the communication overhead due to tasks interdependencies.

A simple way to partition the whole work into different tasks is to assign a fixed number of agents to each available worker [18]. This approach named *agents partitioning* allows a balanced workload but introduce a significant communication overhead (an all–to–all communication is required).

By noticing that actual implementations rely on the fact that the behavior model is designed to mimic natural-inspired models where the limited visibility of real birds allow to bound the range of interaction, most of agent-based simulations systems limit the interaction between agents to a fixed range that is agent's Area of Interest (AOI). Using this observation, several *space partitioning* approaches have been proposed [7], [23], [24] in order to reduce the communication overhead.

In this approach, the space to be simulated is partitioned into *regions*. Each region, together with the agents contained are assigned to a worker. Since the AOI radius of an agent is small compared with the size of a region, the communication is limited to local messages (messages between workers, managing neighboring spaces, etc.). On the other hand, since agents can migrate between regions, load imbalance may occur among the workers. To maintain an even distribution a dynamic partition mechanism is needed: during the simulation, the space partition is updated according to the observed density (number of agents) of the regions. Again the dynamic partitioning should be conducted without introducing too much communication. For instance, it may be not reasonable to update the partitioning using a centralized approach, i.e. the master decides which agents are to be migrated to which worker, since it would require an all–to–all communications.

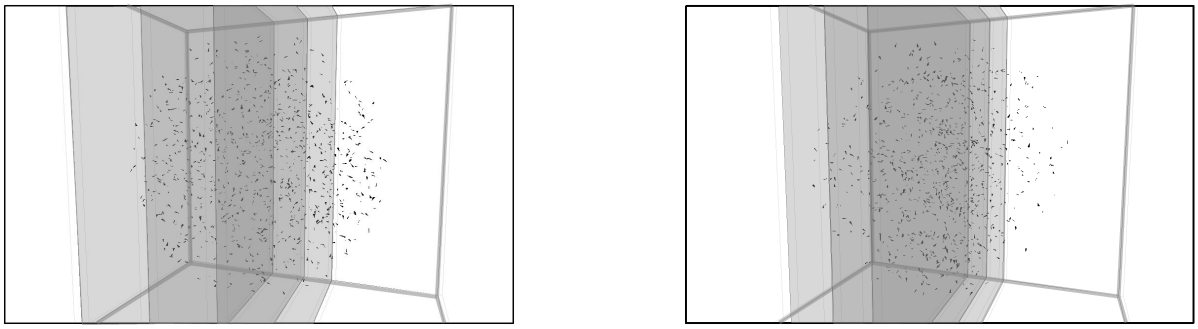

Figure 1. Two snapshots showing $1,000$ agents simulated by 4 workers: (left) a simulation step corresponding to a sparse distribution of agents; (right) a dense distribution (middle regions are thinner). This screenshots have been obtained by gathering simulation data on a single worker that performed the rendering. A video is available [5].

A. Related work

Parallel Agent-based Simulation: While the area of agent-based simulations has been actively investigated for decades, commonly the results are concentrated on small scale simulations, i.e. with few thousands agents. Scientific interest raised in studying large scale simulation, with the interaction of more than $100,000$ agents [21], [23].

Several shared memory agent-based simulation implementations have been proposed, on a large variety of hardware platforms. In [6] the mapping of the flocking behavioral model with obstacles avoidance on streaming-based GPUs is presented. An agent-based simulation optimized for large shared-memory platforms is described in [9]. Similarly, [15] implemented crowds and other flock-like group motion on a multi-core Cell Processor.

Different approaches have been tried on cluster of high performances PCs. Such architectures still require some efforts to tackle the communication/load balancing trade-off. In [12] a 2D parallel framework is proposed that is capable of simulating and rendering the motion of $10,000$ pedestrians in real time. This framework is based on a master/worker paradigm: for each simulation step, the master assigns a portion of pedestrian to each worker. Each worker simulates the pedestrian assigned and sends back the result of its computation.

In [23] a result is presented that is particularly relevant to our discussion but from a different point of view: it present a non-conventional use of the flocking model for document clustering, furthermore they implement such algorithms as an hybrid solution, on a cluster of GPUs.

Dynamic load balancing: Dynamic load balancing schemes represent a challenge for parallel implementation in several context [8]. In general the problem is described by task graph, where nodes represent tasks and edges represent task interactions. Graph and hypergraph partitioning techniques [1] have been employed to schedule tasks onto processors to balance load while taking into account data locality.

In [4] a dynamic portioning schema have been proposed for Parallel Ray Tracing. In [24] the authors implemented Reynolds' model using a space partitioning scheme with centralized load balancing. Unfortunately, the high computational cost of the proposed load balancing schema precludes the use of such approach on each simulation step, as the authors report. Centralized load balancing systems exhibit scalability problem, especially on petascale/exascale systems. In [22] a hierarchical load balancing approach is proposed. The authors argue that decentralized approaches tend to yield poor performances due to incomplete information exchanged by neighboring processor. Other approaches have been explored to parallelize massive simulations on different architectures; for instance in [3] a system is presented that exploits a Peer-to-Peer infrastructure in order to distribute the computational load.

More complex partitioning approaches tackle the load balancing from a geometrical point of view: irregular shape regions (convex hulls) [20]; quad tree, k-d tree, and region growing [17]; Orthogonal Recursive Bisection [7].

B. Our Result

All the works previously cited in this sections exploits centralized or hierarchical load balancing schemes, i.e. involving a worker→master→worker communication pattern every time load balancing is needed. There are two reasons that let us suppose that centralized schema may not be the way to improve significantly the performances, especially in case of large simulations, where all-to-all communication is prohibitively expensive. First, the centralizing communications is bottleneck and may have a negative impact to the system scalability. Second, whereas a centralized balancing schema is used, complex calculations are usually involved, harming system performances.

In this paper we present a novel *distributed load balancing* schema whose purpose is to achieve efficiency *and* an high scalability. Our approach to load balancing is designed to be lightweight and totally distributed: the calculations for the balancing take place at each computational step, and influence the successive step. To the best of our knowledge, our approach is the first distributed load balancing schema in this context.

We present both the design and the implementation that allowed us to perform a number of experiments, with up-to $1,000,000$ agents, whose results are discussed in the

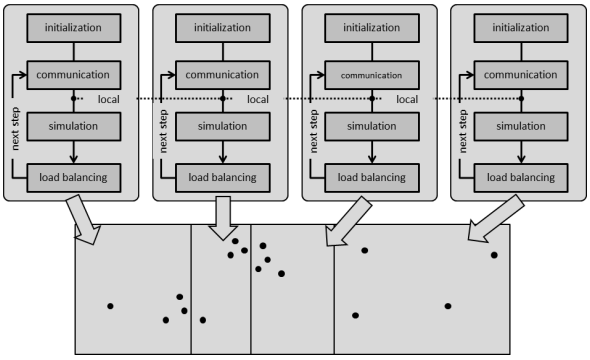

Figure 2. The simulation carried out on 4 workers: upper part the phases each worker executes per step; lower part the main region divided into 4 slices of different sizes, associated to workers.

following sections.

Our results also show that it is not always true that decentralized approaches yield poor performances: the amount of information to be exchanged in order to balance the load depends on the kind of interactions between the tasks. In particular, in Agent-based simulation, the interaction between the task are local and this allows to obtain good performances through a simple local approach.

II. Background

A. Behavior Model

Our work is based on the flocking model developed by Reynolds [16]. Every agent has its own personality that is the result of a weighted sum of a number of behaviors. The simulation is performed in successive steps: at each step, for each agent and for each behavior in the personality, the system calculates a request to accelerate in a certain direction in the space, and sums up all of these requests; then the agent is moved along this result.

The most trivial implementation of the neighborhood calculation consists in a $O(n^2)$ proximity screening, and for this reason the efficiency of the implementation is yet to be considered an issue.

B. Parallel Agent Simulation

We developed a distributed agent-based simulation in order to evaluate and compare the performances of several fully distributed load balancing schemes (cf. Sec. IV).

We use a space partitioning model where each worker maintains a portion of the simulated space, and is responsible for the simulation of agents belonging to such region. In order to guarantee the consistency of parallel implementation with respect to the sequential one, each worker needs to collect information about the neighboring regions.

At the beginning of the simulation, the system randomly generates a quantity of agents within the main region. Each simulation step is formed by three phases (cf. Fig. 2). We describe here the execution of a single step for a single worker. First of all the worker sends to its neighbors the information about the agents belonging its region but that also may fall into the AOI of the neighbor's agents. This information exchange is locally synchronized in order to let the simulation run consistently.

We use a standard approach to achieve a consistent synchronization of the distributed simulations. Each step is associated with a fixed state of the simulation. Regions are simulated step by step. Since the step i of region r is computed by using the states $i-1$ of r's neighborhood, the step i of a region cannot be executed until the states $i-1$ of its neighborhood have been computed and delivered. In other words, each region is synchronized with its neighborhood before each simulation phase.

During the simulation phase the contribution of each behavior for each agent is calculated as a weighted sum. At the end of simulation phase, each worker is able to yield some statistics on the distribution of the agents within the region. These statistics are shared with neighbors workers in such a way that all the workers are able to calculate the novel partitioning on their own. We emphasize that the load balancing algorithm, which moves the boundary between neighbor regions, is quite simple (cf. Sec.IV) and fully distributed, hence it will not represent a bottleneck for the system.

III. Agents partitioning

In order to better exploit the computing power provided by the workers of the system, it is necessary to design the system so that the simulation always evolves in parallel, avoiding bottlenecks. Since the simulation is synchronized after each step, the whole simulation advances with the same speed provided by the slower worker in the system. For this reason it is necessary to design the system in order to balance the load between the workers.

The whole simulation will be carried out in a tridimensional space that will be partitioned along one single dimension in regions (cf. Fig. 1). Depending on the kind of simulation, we define a number of parameters that will influence the load balancing schema. A *main region* will be set large enough to contain all the agents in each step of the simulation; this to assure that agents will not move outside this region. The radius for the agents' AOI is named ϵ. This radius, as well as the shape of the AOI, is correlated to the type of simulation. In the following we assume that ϵ is small compared to the size of a region and each agent, in a single step of simulation, is not allowed to move for more than ϵ. The value of ϵ is important because it will influence the amount of communication between two contiguous workers (i.e., workers with adjacent regions).

A. Handling the boundary

For sake of clarity we will discuss the rationale behind the design of our load balancing schema by describing it in the simpler case of two workers. In Sec. IV-E we will generalize the schema to a any number of workers.

The main region is partitioned into two regions, S_l and S_r. In Fig. 3 we depict this situation, please note that even

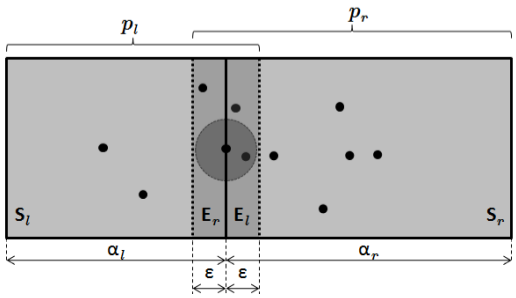

Figure 3. Load balancing with two workers: the main region is partitioned into S_l, simulated by p_l and S_r simulated by p_r. E_l (resp. E_r) represents the portion of S_r, (resp. S_l) that needs to exchange information before each simulation step.

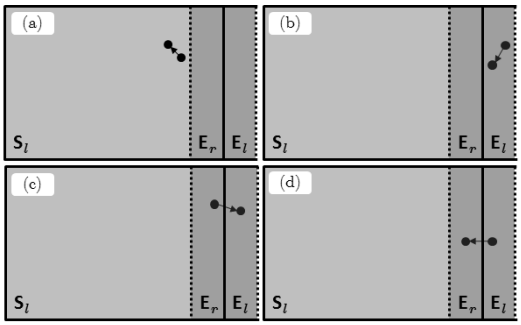

Figure 4. Four cases for agent position when moving to a new simulation step.

if our system is designed for tridimensional space, for sake of clarity, our figures will present a bidimensional space. For brevity, we will describe the idea for worker p_l, without loss of generality. The agents present in the region S_l are simulated by worker p_l. AOI of some of the agents in S_l will intersect S_r and, for this reason, throughout the simulation it will be necessary to share the information about the agents in such AOI. To handle this situation we define E_l as the portion of S_r that contains the agents laying in the AOI of some of the agents in S_l. In other words, E_l is the leftmost ϵ-wide slice of S_r.

The worker p_l, to carry out the simulation of agents in S_l, needs all the agents lying in both S_l and E_l. Before each simulation phase, the position of the agents lying in E_l needs to be updated with information coming from p_r, the same happens for E_r and p_l.

When more than two workers are involved, each worker, except for the first and the last one, has two neighbors. In this case before each simulation phase, each worker communicates with both its neighbor in order to be updated about the position of agents close to its boundaries.

Algorithm 1 Handling the boundary (code for worker p_l)

1: {Partition agent's set in 4 subset}
2: $M_a \longleftarrow$ { agents in case (a) }
3: $M_c \longleftarrow$ { agents in case (c) }
4: $O_c \longleftarrow$ { agents in M_a belonging to E_r }
5: $O_a \longleftarrow M_c$
6: *send* O_a and O_c to p_r
7: *receive* I_a and I_c from p_r
8: agents in $S_l \longleftarrow \{M_a \cup I_a\}$
9: agents in $E_l \longleftarrow \{M_c \cup I_c\}$

B. Special Cases

The exchange of information between p_r and p_l needs to take into account the fact that agents may move across the regions and different cases may raise (cf. Fig. 4): (a) an agent laying in S_l before and after the simulation step; (b) an agent laying in E_l before and after the simulation step; (c) an agent moves from S_l to E_l; (d) an agent moves from E_l to S_l. Case (a) and (b) are easy to be handled because the agent continues to be simulated by the same worker. Cases (c) and (d) is where communication between p_l and p_r is needed to hand over the agent.

We will shortly discuss case (c) in Algorithm 1 where an agent moves from S_l to E_l, in this case two things must be taken into account: the agent will be sent to p_r (line 6) and it must be also kept by p_l because it may belong to the AOI of some of the agents in S_l (line 9).

IV. Distributed Load Balancing Schemes

In this Section we present three load balancing schemes we have developed and tested on our system. The rationale of these schemes is to provide a distributed load balancing by using just local communication.

As described above, the main region is partitioned into regions by slicing it along one single dimension (cf. Fig. 3). We denote with α_l the size of S_l along such dimension. Let s_l (resp. s_r, e_l, e_r) be the number of agents in S_l (resp. S_r, E_l, E_r). Based on the load observed by p_l and its neighbor, the load balancing algorithm will modify the value of α_l by moving the boundary; the purpose is to improve the load balancing for the successive step of computation.

More formally, the load balancing algorithm aims to select the best value of α_l in order to:

1) minimize the unbalancing, that is $|s_l - s_r|$;
2) minimize the communication required for the synchronization phase, that is $|e_l + e_r|$

To apply each one of the following load balancing schema, the workers, p_l and p_r, need some additional information that will be exchanged during the load balancing phase. We will shortly discuss this *overhead* for each of the following algorithms.

Assumptions: Our load balancing schema relays on two assumptions: (i) the measure of the computational load of each worker is linear in the number of agents (ii) the agents are uniformly distributed along the dimension the splitting occurs. The effect of the first approximation will be mitigated by successive refinements of the method, the effects of the second approximation deserve further investigations but we emphasize that, regardless of the effect of this assumption, we experienced good performances also with a large number of agents (cf. Sec. V).

A. Static partitioning (static)

In order to provide a baseline scheme which will be compared with our proposals, we have implemented a static partitioning, where the value of α is fixed, for each worker, to d_x/w, where w is the number of workers/regions and d_x is the size of the main region along the splitting dimension. The scheme will be also used to evaluate the degree of unbalancing of a given testbed simulation.

B. Region wide load balancing (dynamic1)

Assuming that agents are uniformly distributed along the splitting dimension in the whole region assigned to a worker, we may define a simple algorithm that moves the boundary by evaluating the values of s_l and s_r.

Let $\alpha_l(t)$ be the value of α_l at simulation step t. The load balancing algorithm updates the value of α_l according to the following equation:

$$\alpha_l(t+1) = \alpha_l(t) + \frac{s_r - s_l}{2} \cdot \frac{\alpha_l(t) + \alpha_r(t)}{s_l + s_r}, \qquad (1)$$

where the first fraction represents the number of agents to be moved to balance the load between p_l and p_r and the second one is the amount of linear space containing a single agent, under the "uniformly distributed agents" assumption stated above.

The overhead of communication, for p_l, is the transmission of s_r and α_r. Thus, after each simulation phase, p_l exchanges such information with p_r and then both, use Eq. 1 to compute the new partitioning. Notice that no additional communication is required to spread the updated boundary position. Clearly this communication does not represent a bottleneck because it is limited to only two values, for each step. Notice that, when the number of worker is 2, the exchange of information is not strictly required because both s_r and α_r can be calculate by p_l by using s_l and α_l, respectively. Of course this is not true in the most general case with more workers.

C. Mitigated region wide load balancing (dynamic2)

In this algorithm we aim at mitigate the effects of approximation of considering agents uniformly distributed across the space. In general, such distribution depends on the behavioral model and is usually not uniform. For instance several flock simulations models converge to a state where agents aggregate into few dense groups [2]. On the other hand, if the distribution had been uniform, a static partitioning would be enough to achieve a good load balancing.

The load balancing schema above is prone to a phenomenon, when a large flock of agents rapidly moves between the two workers: the boundary is moved too aggressively in the attempt of balancing the flock movement (cf. Sec. V for test results) and it oscillates. Such oscillation is an effect of the assumption of uniformity of distribution, that underestimates the amount of agents approaching the boundary. Clearly the wider this oscillation is the larger is the number of agents to be handed over between p_l and p_r.

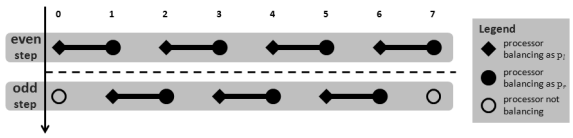

Figure 5. Load balancing with multiple workers. Each worker, after each simulation step, updates, by turns, one of is boundaries. Turns are chosen in such a way that neighbor workers always update the position of their shared boundary at the same simulation steps.

To mitigate this effect we have slightly changed the balancing equation, by adding a constant k that adds some inertia in moving the boundary:

$$\alpha_l(t+1) = \alpha_l(t) + k \cdot \frac{s_r - s_l}{2} \cdot \frac{\alpha_l(t) + \alpha_r(t)}{s_l + s_r} \qquad (2)$$

We experimentally observed that this version of the equation with $k = 1/2$ provides good results in alleviating the oscillations.

D. Restricted assumption load balancing (dynamic3)

We refined the algorithm even more by relaxing the assumption of uniformity of the distribution of the agents: the assumption will be applied just to the spaces E_l and E_r, instead of S_l and S_r. As a counterpart we had to limit the per-step movement of the boundary to ϵ which represent the size of both E_l and E_r along the splitting dimension. However such restriction does not represent a real limitation since during the test we have performed, the requested movement was always smaller than ϵ.

$$\alpha_l(t+1) = \alpha_l(t) + \begin{cases} \min\left(\epsilon, \frac{s_r - s_l}{2} \cdot \frac{\epsilon}{e_l}\right) & \text{if } s_r > s_l, \\ \max\left(-\epsilon, \frac{s_r - s_l}{2} \cdot \frac{\epsilon}{e_r}\right) & \text{otherwise.} \end{cases}$$

In this new equation the overhead of communication consists in exchanging the values of s_l, e_l and α_l. Again such information can be rapidly shared between neighbor workers.

E. Generalization to multiple workers

The load balancing schemes we defined above for two workers can be easily generalized to any number of workers. We describe here a distributed load balancing schema for parallel agent based simulations. The system is composed by a set of n workers having a linear topology, i.e. worker p_i has two neighbors, p_{i-1} and p_{i+1}. Obviously, p_0 and p_{n-1} have a single neighbor.

The rationale behind the generalization is straightforward, in the 2-workers load balancing schema we used p_l and p_r to indicate the two workers. The workers will be distinguished, by their index, in *even* workers and *odd* workers. The idea is to apply the same 2-workers schema to couples of neighbors workers, on alternate steps: on even simulation steps, even workers play the role of p_l and the odd workers play the role of p_r, while in the odd simulation steps even workers will be p_r and odd workers will be p_l. In Fig. 5 is depicted the generalization in the case of 8 workers: two successive steps of simulation are shown. In the visualization it appears

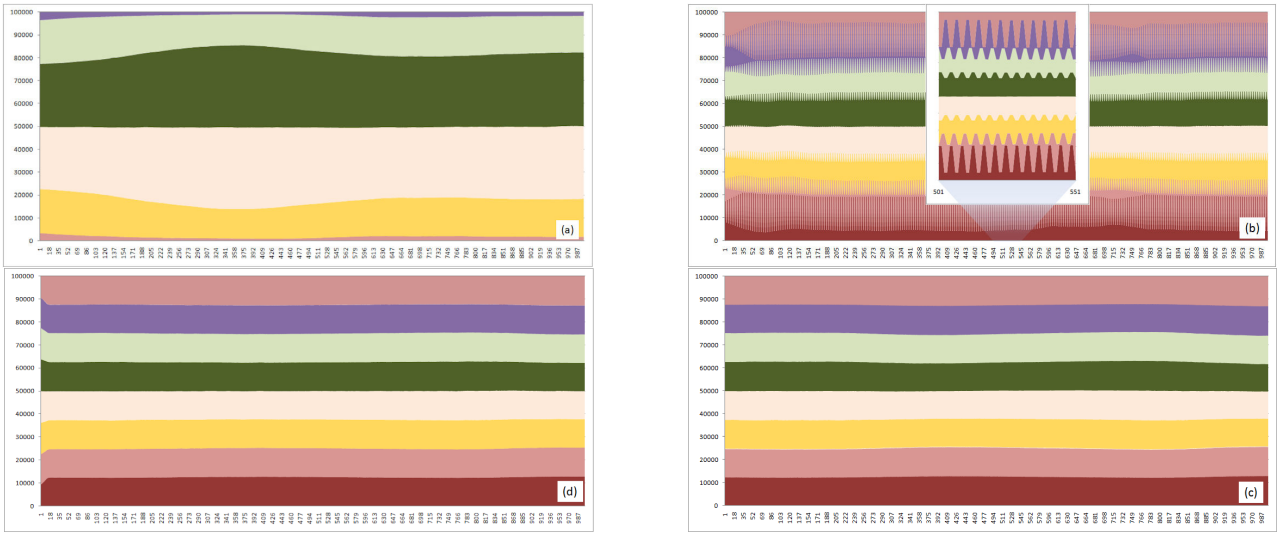

Figure 6. Distribution of agents per worker. Each color represents a different worker. x-axis indicates the simulation step ($1,000$ simulation steps are depicted) while th y-axis represents the distribution of agents. (a) $(100,000;\ 8;\ \textit{static})$ (b) $(100,000;\ 8;\ \textit{dynamic}1)$ (c) $(100,000;\ 8;\ \textit{dynamic}2)$ and (d) $(100,000;\ 8;\ \textit{dynamic}3)$.

clear how on alternate steps worker 0 and worker n will not perform any load balancing.

V. Tests and performances

A. Test setting

We have implemented the 4 methods by developing a parallel version of OpenSteer[14]. OpenSteer is an open-source C++ library that implements a plurality of steering behaviors to be used as a standard library for videogames.

Hardware details: Test machine is an IBM HS21 Cluster with 256 nodes available at CRESCO Project computing platform, Portici ENEA Center. Each node is equipped with 2 Xeon Quad-Core Clovertown E5345 at 2.33 GHz and 16 GByte RAM. The nodes are interconnected with an Infiniband network.

Software details: Our parallelization is based on MPI [10]. In particular the system mapped MPI processes onto cores. Underlying MPI implementation implicitly switches between most suitable protocol to let workers to communicate (e.g. Infiniband rather than Inter-Processes Communication).

We performed tests using a different number of agents, but fixed agent density (hence setting the main region volume accordingly). We set a *bounding radius* in order to assure that agents does not go outside the main region. For each agent overtaking this radius, an additional backward steering force is added to the standard model.

The state of an agent comprises two vectors: position and speed. Other common properties (i.e. mass) are defined constant and are not shared/sent between workers. Of course, the bigger is the state of an agent, the more expensive will be the communication overhead of the parallelization.

B. Load balancing analysis

The batch of tests we performed simulates $1,000$, $10,000$, $100,000$ agents running on 8, 16, 32 and 64 cores. Each tests lasts $11,000$ simulation steps, the first $1,000$ steps have been discarded in order to let the simulation to stabilize. The set of tests we performed is the result of the Cartesian product $\{1,000,\ 10,000,\ 100,000\} \times \{8,\ 16,\ 32,\ 64\} \times \{\textit{static},\ \textit{dynamic}1,\ \textit{dynamic}2,\ \textit{dynamic}3\}$. In the next paragraphs and in figures we will indicate the test setting by using a triple took from such set.

For each simulation step run, we collected the number of agents simulated by each worker/core and the number of agents exchanged between neighbor workers (communication).

The test results are encouraging and confirm that *dynamic*2 and *dynamic*3 handle the balancing of the agents between neighbor workers pretty well. Moreover, the amount of communication overhead injected by the hand over of the agents between neighbor workers is negligible. To avoid cluttering we illustrate the $(100,000;\ 8;\ *)$ cases in Fig. 6, reporting the distribution of agents among workers, in each simulation step, from upper left and clock-wise we have: *static, dynamic*1*, dynamic*2*, dynamic*3. In Fig. 6.(a) $(100,000;\ 8;\ \textit{static})$ is shown and it depicts the heavy unbalancing in the distribution of the load: two of the workers simulate an average of $\approx 30,000$ agents instead of the ideal $12,500$. Fig. 6.(b) shows the *dynamic*1 algorithm which provides a better balancing but suffers of the oscillation phenomenon mentioned in Sec. IV-C. To ameliorate the visibility of such phenomenon we provides a zoomed section of the graph (50 simulation steps). The two successive algorithms, *dynamic*2 and *dynamic*3, are shown in Fig. 6.(c) and (d), respectively. The figures represent an almost optimal balanced graph showing that each worker handles $\approx 12,500$ agents. Both the algorithms sensibly reduce the oscillations, even if they are still measurable, shown in Fig. 7; please note how *dynamic*3 (b) behaves slightly better than *dynamic*2 (a),

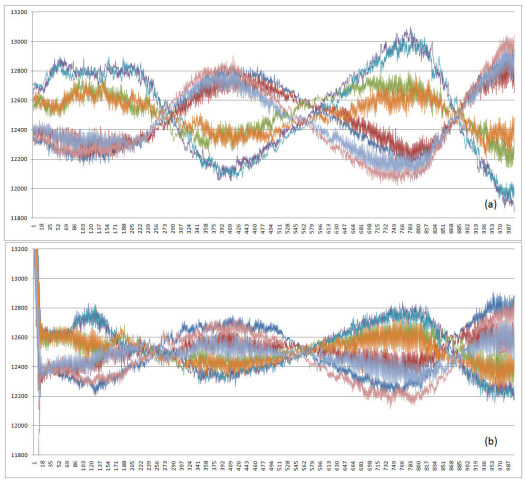

Figure 7. Number of agents per worker: each series indicates the number of agents per worker. x-axis indicates the simulation step ($1,000$ simulation step are depicted) while the y-axis represents the number of agents. (a) $(100,000;\ 8;\ dynamic2)$. (b) $(100,000;\ 8;\ dynamic3)$.

by providing oscillations that have smaller amplitude.

In Table I we summarize the results we measured in other test settings. For each test we report the standard deviation (σ) of the number of agents per worker and the total number of agents exchanged during the communication phase of the algorithm (communication). The results indicates that together with a good performance in reducing the unbalancing we measure an increase in the communication cost. The best refinement of the load balancing schema, *dynamic*3 provides substantially smaller standard deviation but needed more than the double of agents exchange between workers, respect to the static partitioning. The oscillations we noted in *dynamic*1 deeply impact on the communication cost, this can be noticed by comparing the performances of $(100,000;\ 8/16;\ dynamic1)$ and $(100,000;\ 8/16;\ dynamic2/dynamic3)$. On the other hand, when the number of workers is higher, *dynamic*1 behaves as *dynamic*2 and *dynamic*3 in terms of communication but the balancing worsens. Overall the *dynamic*3 algorithm performs pretty well on all test cases. Moreover, the improvement provided by dynamic algorithms increase as the number of either workers or agents grow (see scalability in Fig. 8).

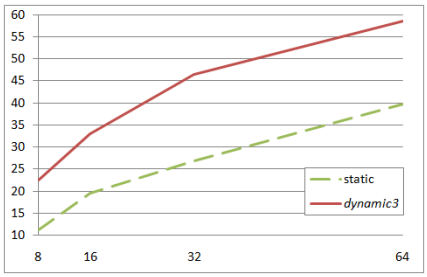

Figure 8. Scalability: A comparison between $(100,000;*;\ static)$ (dotted line) and $(100,000;*;\ dynamic3)$ (continuous line). For each algorithm, $100,000$ agents are simulated with $8, 16, 32$ and 64 workers. Each tests lasts $12,000$ simulation steps. x-axis indicates the number of workers; y-axis plots the average number of simulation steps per second.

Discussion: Finally, we performed a quite massive simulation: $(1,000,000;\ 64;\ dynamic3)$. The objective of this test is to measure the system performances in circumstances that can hardly be managed by a single machine. The execution of a run of $2,000$ steps required[1] $140,754$ ms, which correspond to 14.21 simulation steps computed per second. We have also observed that, after a small number of steps (around $1,000$ on the test with $1,000,000$ agents), where the load balancing algorithm stabilizes, the performances of the system are quite constant.

It is worth to mention that the load balancing strategy provides also several additional benefits. For instance, it allows to simplify the tuning of the spatial data structure (e.g. select the appropriate grid resolution) which is used for proximity screening. Since the optimal granularity of the spatial data structure depends on the number of agents to be managed, when the workload balance is assured, we may assume that the number of agent per worker is roughly constant, hence we are able to choose the optimal granularity.

VI. Conclusion

Agent based simulations, due to their computational power requirements, appear to be a natural application for parallel architectures. In this context it is challenging to design the system so that the simulation evolves in parallel, avoiding bottlenecks, in order to better exploit such computing power. Since the simulation must be synchronized after each step, the system advances with the same speed as the slower worker in the system is capable of. For this reason it is necessary to take into account a good implementation of a load balancing mechanism. Several centralized load balancing schemes have been proposed. A common problem with these approaches is that the centralized management usually requires a large amount of communication – between workers and the master node, which act as a load balancing manager – that consumes bandwidth and introduces latency.

We presented a novel *distributed load balancing* schema whose purpose is to achieve an effective load balancing introducing a low communication overhead. Our schema is designed to be lightweight and totally distributed: the calculations for the balancing take place on every worker at each computational step, and influences the successive step; each communication is local (i.e., between neighbor workers). To the best of our knowledge, our approach is the first distributed load balancing schema in this context.

We presented both the design and the implementation that allowed us to perform a number of experiments, with up-to $1,000,000$ agents. The tests revealed that the architecture presents a quite good scalability: the communication overhead, due to the local workers interaction, is dominated by

[1]In order to measure the total parallel computation time, we used a specific worker that, on every step, collects information about the global completion time.

Table I
LOAD BALANCING/COMMUNICATION RESULTS

×	(*static*)	(*dynamic*1)	(*dynamic*2)	(*dynamic*3)
$(1,000; 8)\{\sigma$ - *communication*$\}$	131.16 - 272,976	36.03 - 624,562	12.44 - 573,275	6.39 - 584,961
$(10,000; 8)\{\sigma$ - *communication*$\}$	1292.54 - 581,611	550.83 - 2,336,951	13.44 - 1,355,381	6.49 - 1,236,951
$(100,000; 8)\{\sigma$ - *communication*$\}$	11,480.80 - 746,753	5,324.28 - 22,261,600	10.94 - 1,278,788	5.14 - 1,217,074
$(100,000; 16)\{\sigma$ - *communication*$\}$	5,815.35 - 1,407,684	2,146.36 - 16,067,865	12.13 - 3,504,703	7.43 - 3,484,038
$(100,000; 32)\{\sigma$ - *communication*$\}$	3,945.53 - 2,927,194	743.21 - 12,714,472	30.55 - 6,896,584	9.12 - 6,877,622
$(100,000; 64)\{\sigma$ - *communication*$\}$	1,975.22 - 5,622,336	260.21 - 15,888,243	106.06 - 13,679,714	19.44 - 13,204,903

the speed-up achieved thanks to the better load balancing, provided by our schema.

Future works: Our load balancing schema aims at balancing along a single dimension the uneven distribution of agents in a tridimensional space. A reasonable evolution of such schema is to take into account the fact that the space is 3d: current implementation does not properly balance work when agent clustering fully exploits the three dimensions. For instance, in the simulation of a flock of birds or a school of fishes we may find several flocks overlapping and spreading across the whole space and not just lying along one single dimension. We plan to extend our technique to a multi dimensional space.

We reported some early tests of a simulation with $1,000,000$ agents. One technical problem we solved was the creation of such amount of agents: this phase is still centralized, in the current version of the system, and for this reason the number of agents in the system was limited by the memory (and the capability of representation) of a single worker. We plan to distribute such phase in order to reach a number of agents that is proportional to the number of workers in the system and it is clear that in such scenario we would easily reach the goal of a multi-millions agents simulation.

REFERENCES

[1] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st Intl. Par. and Distr. Proc. Symp. (IPDPS)*, pages 1–11. IEEE, 2007.

[2] B. Chazelle. Natural algorithms. In *Proc. of the 20th ACM-SIAM Symp. on Discr. Alg. (SODA'09)*, pages 422–431, 2009.

[3] G. Cordasco, R. De Chiara, U. Erra, and V. Scarano. Some Considerations on the Design of a P2P Infrastructure for Massive Simulations. In *Proc. of Inter. Conf. on Ultra Modern Telecommunications (ICUMT '09)*, 2009.

[4] G. Cordasco, B. Cosenza, R. De Chiara, U. Erra, and V. Scarano. Experiences with Mesh-like computations using Prediction Binary Trees. *Scalable Computing: Practice and Experience, Scientific international journal for parallel and distributed computing (SCPE)*, 10(2):173–187, June 2009.

[5] B. Cosenza, G. Cordasco, R. De Chiara and V. Scarano. http://www.isislab.it/projects/DistrSteer/ Dist Steer: Parallel Distributed Agent-Based Simulations, 2010.

[6] R. De Chiara, U. Erra, V. Scarano and M. Tatafiore. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In *Proceedings of Vision, Modeling and Visualization 2004 (VMV)*, Nov. 2004.

[7] F. Fleissner and P. Eberhard. Load Balanced Parallel Simulation of Particle-Fluid DEM-SPH Systems with Moving Boundaries. In *Proc. of Parallel Computing: Architectures, Algorithms and Applications (ParCo'07)*, 2007.

[8] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, Inc., 1998.

[9] B. Knafla and C. Leopold. Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP. In *Proc. of Parallel Computing: Architectures, Algorithms and Applications (ParCo'07)*, 2007.

[10] The Message Passing Interface (MPI) standard. http://www.mpi-forum.org/

[11] R. Narain, A. Golas, S. Curtis, and M.C. Lin. Aggregate dynamics for dense crowd simulation. In *ACM SIGGRAPH Asia 2009 papers*, pages 1–8, New York, NY, USA, 2009.

[12] M. J. Quinn, R. A. Metoyer, and K. Hunterzaworski. Parallel implementation of the social forces model. In *in Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics*, pages 63–74, 2003.

[13] W. T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 359–375, New York, NY, USA, 1983. ACM.

[14] OpenSteer, Steering Behaviors for Autonomous Characters. http://opensteer.sourceforge.net/, 2004

[15] C. W. Reynolds. Big fast crowds on ps3. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121, New York, NY, USA, 2006. ACM.

[16] C. 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.

[17] A. Steed, and R. Abou-haidar. Partitioning crowded virtual environments. In *In Proceedings of the ACM symposium on Virtual reality software and technology*, pages 7–14, 2003.

[18] S. Plimpton. Fast parallel algorithms for short range molecular dynamics. *Journal of Computational Physics*, 117 n.1:1–19, 1995.

[19] A. Treuille, S. Cooper, and Z. Popović. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1160–1168, New York, NY, USA, 2006. ACM.

[20] G. Vigueras, M. Lozano, J. M. Orduna, and F. Grimaldo. A comparative study of partitioning methods for crowd simulations. *Appl. Soft Comput.*, 10(1):225–235, 2010.

[21] G. Yamamoto, H. Tai, and H. Mizuta. A platform for massive agent-based simulation and its evaluation. In *Proc. 6th Intern. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, 2007.

[22] G. Zheng, E. Meneses, A. Bhatelé and L. V. Kalé Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, 2010.

[23] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Large-Scale Multi-Dimensional Document Clustering on GPU Clusters. In *IEEE International Parallel and Distributed Processing Symposium*, 2010.

[24] B. Zhou and S. Zhou. Parallel simulation of group behaviors. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 364–370. Winter Simulation Conference, 2004.