The Journal of Supercomputing (2019) 75:1524–1550
https://doi.org/10.1007/s11227-018-2688-8

Designing a benchmark for the performance evaluation of agent-based simulation applications on HPC

Andreu Moreno[1,2] · Juan J. Rodríguez[3] · Daniel Beltrán[2] · Anna Sikora[2] · Josep Jorba[3] · Eduardo César[2]

Published online: 17 November 2018

Abstract
Agent-based modeling and simulation (ABMS) is a class of computational models for simulating the actions and interactions of autonomous agents with the goal of assessing their effects on a system as a whole. Several frameworks for generating parallel ABMS applications have been developed taking advantage of their common characteristics, but there is a lack of a general benchmark for comparing the performance of the generated applications. We propose and design a benchmark that takes into consideration the most common characteristics of this type of applications and includes parameters for influencing their relevant performance aspects. We provide an initial implementation of the benchmark for FLAME, FLAME GPU, Repast HPC and EcoLab, some of the most popular parallel ABMS platforms, and use it for comparing the applications generated by these platforms. The obtained results are mostly in agreement with previous studies, but the designed and implemented specification has allowed for testing a wider set of aspects, such as the number of interacting agents, the amount of interchanged data or the evolution of the workload and obtaining more reliable results.

Keywords Agent-based modeling and simulation · Parallel applications · Performance · Benchmark

1 Introduction

Agent-based modeling and simulation (ABMS) is a class of computational models for simulating the actions and interactions of autonomous agents (both individual and collective entities such as organizations or groups) with the goal of assessing their

This work has the support of the Ministerio de Economía, Industria y Competitividad MINECO-SPAIN under contracts TIN2014-53234-C2-1-R and TIN2017-84553-C2-1-R and by the Generalitat de Catalunya GenCat-DIUiE (GRR) with the Project 2017-SGR-313.

Extended author information available on the last page of the article effects on a system as a whole [1]. ABMS belongs to a category of models known as discrete event simulations, which run with some set of starting conditions over some period of time, allowing the programmed agents to carry out their actions until some specified stopping criterion is satisfied, usually either a certain amount of time or a specified system state [2].

An agent is an autonomous, dynamic rule-based entity within a defined environment. The behavior of the agents is encoded in algorithms, which may go from simple deterministic rules to sophisticated algorithms including learning and adaptive strategies. The agents determine the dynamics of the system as a whole by interacting with each other. Being able to communicate with each other, the agents can influence the behavior of other agents creating complex interactions within a system.

Depending on the complexity of the model and the number of agents participating in the simulation, an ABMS application may consume a significant amount of computational resources. Consequently, in many cases these simulations can take advantage of parallel techniques and HPC hardware. In addition, parallelizing this type of systems is feasible due to the underlying autonomous behavior of the agents, taking into account the related synchronization and communication issues.

Several frameworks for generating parallel ABMS simulators have been developed taking advantage of the common characteristics of these applications. Among them, we can mention FLAME [3], FLAME GPU [4], Repast HPC [5], EcoLab [6], D-MASON [7], Pandora [8] or Care-HPS [9]. They present differences in the way agents and contexts are specified (e.g., C, C++, Java, XML, Tcl), in the way communications are managed (e.g., explicit messages, agent replication), in the presence of a load balancing mechanism, in the performance of the resulting simulator, etc. Consequently, how can a potentially non-expert user decide which is the best framework to implement a given agent-based model?

There are some comparative studies, such as [10,11] or [12], that can be very helpful for taking this decision because they take into consideration aspects like development effort, framework features and performance. However, only a few studies are focused on parallel/distributed platforms, and sometimes, comparison criteria are subjective; for example, development effort usually depends on the developers programming background.

Moreover, as noticed by Open Agent Benchmark Initiative for Parallel and Distributed Benchmarking (OpenAB) [13], in the case of empirical studies, there is an urgent need of a comprehensive common base application that allows for a fair platform comparison. Benchmarking is the quantitative foundation of the software or hardware devices developed in most research areas.

For this reason, defining a common benchmark for comparing the performance of parallel ABMS generation platforms is the main motivation behind this work. We are presenting the design of a benchmark that takes into consideration the model's computation load and distribution, communication load and pattern, and problem size. The proposed benchmark is a generalization of the one already used in [11]. An initial implementation of the benchmark has been developed in Repast HPC, FLAME, FLAME GPU and EcoLab.

These implementations have been used to conduct a set of comparison experiments. Part of these experiments reproduces the study presented in [11]; the obtained results are mostly coincident, but in several cases the new results extend and correct the original ones. The rest of the experiments shows new results for all the considered platforms.

The remainder of this work is organized as follows. Section 2 summarizes the ABMS frameworks used for implementing and testing the benchmark. Next, Sect. 3 discusses the decisions taken in the design and implementation of the proposed benchmark. Section 4 depicts the use of the benchmark for comparing well-known ABMS frameworks, FLAME, FLAME GPU, Repast HPC and EcoLab. Section 5 describes the related work. Finally, conclusions are stated in Sect. 6.

2 Background

This work is focused on the design of a benchmark for comparing the performance of the parallel simulators generated by different frameworks, and logically, we need to test it using representative cases of study. In this section, we introduce a summary of the most relevant characteristics of FLAME, FLAME GPU, EcoLab and Repast HPC, which are the frameworks that have been used in these cases of study. We have focused on these platforms because all of them are backed by several publications, have been used for implementing real applications, offer stable and maintained versions of the software, and have been specifically designed for HPC systems.

2.1 FLAME

FLAME [3] was developed at the University of Sheffield in collaboration with the Science and Technology Facilities Council (STFC). FLAME has been used to solve problems involving multiple domains such as economical, medical, biological and social sciences. This framework allows for writing several agents models using a common simulation environment and then performing simulations in a simple way on different parallel architectures.

FLAME is a tool able to generate the necessary source code for the simulation. It automatically generates the simulation code in C through a template engine.

The model specification is described by two types of files: XMML (X-Machine Markup Language) files, which is a dialect of XML, and the implementation of the agent functions contained in C files. This approach is similar to the one followed by Repast [5] and Pandora [8], but in this case, instead of using an object oriented language, agents state and data are specified using XMML.

The functionality of FLAME is based on finite-state machines called X-machines, which consists of a finite set of states, transitions between states, messages between agents and actions. To perform the simulation, FLAME holds each agent as an X-machine data structure, whose state is changed via a set of transition functions. Furthermore, transition functions may perform message exchanges between agents.

The transitions between the states of the agents are accomplished by keeping the X-machines in linked lists. The simulation environment has one linked list for each state of a specific kind of agent. During the simulation, all agents' X-machines are inserted into

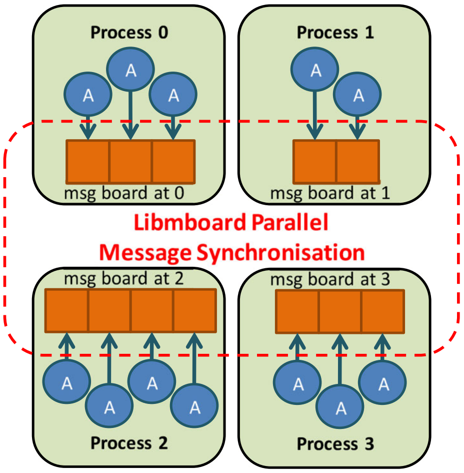


Fig. 1 Parallel communication and synchronization in FLAME via libmboard

the list associated with their initial state. Next, the corresponding transition function is applied to each X-machine, and they are moved to the list associated with the agents' next state. This process is repeated until all agents reach the last state, which determines the end of the iteration.

When FLAME generates parallel code, this structure is replicated in all the nodes participating in the simulation and the agents are distributed among these nodes. In addition, a communication library called *libmboard*, which is build on MPI, is used for managing communication between agents assigned to different nodes. This library sends all messages to external agents through a coordinated communication mechanism between different MPI processes. Figure 1 schematically shows how communication takes place in FLAME.

In this way, FLAME provides a general communication mechanism that allows any pair of agents to interchange messages without needing any replication of agents in different nodes.

Distribution of agents among nodes in FLAME is based on two static partitioning methods: geometric and round-robin partitioning.

FLAME also has some drawbacks that do not depend on the particular model being defined and simulated. First, its current version does not include any mechanism to enable the movement of agents between processes, and second, the centralized communication scheme based on libmboard limits the scalability of the generated simulators.

2.1.1 Extended FLAME

A set of works have proposed a methodology that enables dynamic and automatic performance enhancements for FLAME-generated simulators implementing spatially explicit ABMS [14–16]. The methodology introduces a tuning strategy which dynamically minimizes the gaps of the computing and communication workloads between simulation processes. The strategy adjusts the global simulation workload migrating groups of agents among the simulation process according to their computation workload and their interconnectivity modeled using a hypergraph. A hypergraph is a graph generalization that, in this case, allows for more accurately modeling agent system interactions. This hypergraph is lastly partitioned using a parallel partitioning algorithm, implemented in Zoltan [17], to decide a proper workload distribution.

This strategy is divided in two phases: monitoring and tuning. In the monitoring phase, the application workload is measured at runtime in order to identify load unbalances and when necessary applying the load balancing strategy in the tuning phase. The load balancing strategy consists of three steps:

1. Create an **agent system representation** (ASR), which uses a clustering algorithm for representing the system agents as a global hypergraph.
2. **Hypergraph partitioning**, using Zoltan, in order to determine a more balanced distribution.
3. **Agent migration** to adjust the global workload.

2.2 FLAME GPU

FLAME GPU [4,18] is an extension of the FLAME framework which provides a mapping between the XMML + C agent specification and CUDA code, producing a parallel simulator for running in NVIDIA GPU devices.

This framework allows developers to focus on specifying agent behavior and run simulations without being concerned about CUDA programming or devising optimization strategies. In addition, simulations performance is usually significantly improved in comparison with the one obtained using only the CPU. This allows for the simulation of larger models with less cost than cluster based simulations.

The FLAME GPU functionality is based on X-machines as in FLAME, with some differences that are specified in the FLAME GPU documentation [19]. Initially, FLAME GPU was developed for Windows platforms, but now there is a GNU/Linux version of the framework.

FLAME GPU is based on a set of XSLT (eXtensible Stylesheet Language Transformations) templates. As it needs an XSLT processor, we are using the XSLT Apache Xalan [20] processor to generate the model code. FLAME GPU also needs the OpenGL Extension Wrangler library [21] and the CUDA Data Parallel Primitives library [22] to run the generated simulation.

The FLAME GPU-generated simulator has two main data structures: one for the agents and the other for the messages. These structures are allocated into memory at execution time and manipulated by CUDA kernels when the simulation is executed. As in FLAME, the agent state is changed by iterating on a function set which makes agents evolve.

The communication between agents is done through messages; however, unlike in the case of FLAME, in FLAME GPU there is message partitioning instead of agent partitioning. Each message is contained in a list and it consists of a name, a description, a list of variables and the partitioning definition. There are three message partitioning types:

1. **Non-partitioned messages** make possible interactions between any pair of agents, as in the case of FLAME with libmboard. This is a very flexible but costly mechanism because when an agent wants to read a message it has to iterate over the whole list of messages.
2. **Discrete partitioned messages** establish the maximum distance in a discrete 2D space between interacting agents. Agents must define variables for their coordinates (x and y), their position cannot change during the simulation, and the message size must be a power of two.
3. **Spatial partitioned messages** establish the maximum distance in a continuous 3D space between interacting agents. Agents must define variables for their coordinates (x, y, z), and their position can change during the simulation. In this case, the model must specify three space bounds that define the space where messages may exist. The space within these bounds is divided in P partitions for each dimension, according to the defined distance in the following way:

   $$P = \lceil((\text{maxbound} - \text{minbound}) \div \text{distance})\rceil$$

   These partitions are used to generate a partition boundary matrix which holds indexes of messages from each partition in the whole space.

FLAME GPU has a restriction because messages can only use simple type variables and graph communication. This is a problem when it is necessary to interchange messages with multiple elements because arrays are not supported and it is necessary to add multiple simple type variables.

2.3 Repast HPC

Repast for HPC (Repast HPC) [5] is an ABMS toolkit for high-performance distributed computing platforms that was released in 2012. It implements the core Repast Simphony [5] concepts and features, and it is written in C++ using MPI for parallel operations.

Agents are implemented as objects written in C++. An agents state is represented by the field variables of those classes and agent behavior by methods in those classes.

The simulation proceeds using a schedule. Repast HPC implements a dynamic discrete event scheduler with conservative synchronization. The user schedules events to occur at a specific tick, and the ticks determine the relative order in which the events occur.

A context is used to encapsulate the population of agents. As agents are created they are added to a context and when they die, they are removed from a context.

The context has projections associated with it. A projection imposes a relational structure on the agents in the context. For example, a grid projection puts agents into a grid matrix type structure where each agent occupies some cell location in the grid. A network projection allows agents to have network link relationships with each other. Repast HPC implements 3 types of projections: grid, continuous space and network.

A Repast HPC simulation is thus composed of a set of agents, one or more contexts containing these agents and zero or more projections relating them. Simulation exe-

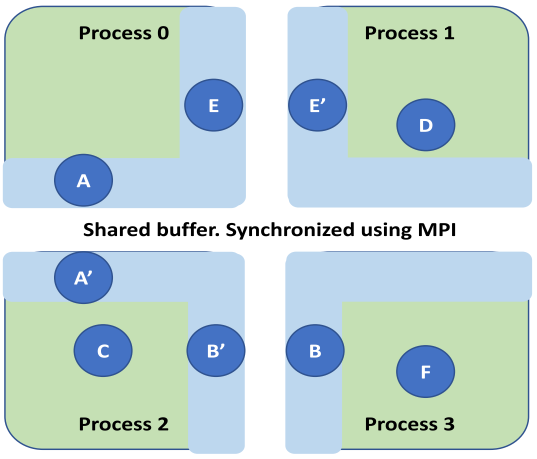


Fig. 2 Parallel communication and synchronization in Repast HPC

cution typically consists of getting the next scheduled event and executing that event invoking some agent's method.

When run in parallel, each process is responsible for executing a set of local agents (those assigned to the process). When agents assigned to a certain process (e.g., agent A at Process 0) interact with other agents assigned to a different process (e.g., agent C at Process 2), the relevant agents in Process 0 are replicated and copied to Process 2 as shown in Fig. 2. Repast HPC employs user-provided methods for serializing/de-serializing the agents and MPI for communicating them.

As in the case of FLAME, Repast HPC has some drawbacks that do not depend on the particular model being defined and simulated. Communication among agents implies replication of objects in several processes, thus incrementing memory requirements, forcing users to explicitly synchronize replicas and making bidirectional communication more difficult.

2.4 EcoLab

EcoLab [6] is an ABMS framework designed for C++ programmers that allows for implementing agent-based models in C++ and simulating them using Tool Command Language (Tcl) [23]. EcoLab supports parallel simulation of agents distributed over an arbitrary topology graph through Graphcode and Classdesc libraries [24].

Similarly to Repast HPC, agents are implemented as objects in C++. Each agent is assigned to a node in a graph that defines the relationships between agents (similarly to Repast HPC context and projections). This graph is managed from a global model class.

The simulation process is implemented through a Tcl script that is responsible for creating the initial set of agents, distributing them in the graph and calling the model methods that perform the actual simulation.

When executed in parallel, each simulation process executes the Tcl script, which means that it creates a subset of the system's agents and is responsible for simulating its local agents. As in the case of Repast HPC, interactions between agents assigned to different processes are managed by replicating the relevant agents in both processes. However, EcoLab automatically provides the serialization/deserialization methods for the agents and agents' containers (Classdesc). A library called Graphcode is used for performing the communication based on MPI, but this library provides more than a communication mechanism; it uses ParMETIS [25] for balancing the load of the whole system.

As in the case of the other frameworks, EcoLab has some drawbacks. Interaction among agents implies replication of objects in several processes, thus incrementing memory requirements, forcing users to take care of synchronization and repartitioning. In addition, the provided load balancing approach is centralized in one process, which is responsible for gathering the current configuration of the agents graph, calling the ParMETIS partitioning function and distributing its results among the other processes.

3 Benchmark description

A Benchmark must be representative of the target applications being assessed. Consequently, the first step for creating a benchmark for parallel ABMS applications should be to define a set of requirements extracted from common ABMS cases. Next, a benchmark must be designed for satisfying the detected requirements, and finally, it shall be implemented on the platforms being assessed.

Therefore, in this section, we present the main contribution of this work, which is covering these three phases for creating a benchmark for assessing parallel ABMS applications.

3.1 Benchmark requirements

In the first place, probably the most relevant features of an ABMS is the possibility of defining interactions between agents because it is the main source of emerging behavior in these systems. Usually, the frameworks for generating parallel ABMS simulators offer some kind of abstraction for allowing programmers to express these interactions at the agent level. For example, FLAME and FLAME GPU offer an explicit message passing mechanism, while Repast HPC and EcoLab allow an agent to get a list of objects representing its neighbors and then call these objects methods.

This abstraction is implemented using a mechanism that allows to communicate and synchronize threads and/or processes on the target hardware system. For example, FLAME, EcoLab and Repast HPC use Message Passing Interface (MPI) [26] for implementing their communication abstractions, while FLAME GPU uses shared memory for communicating agents. Logically, any communication and synchronization mechanism used in a parallel application introduces an overhead, which depends on the parallel programming model (shared memory, message passing). This overhead must be measured for assessing the application's performance.

In most ABMS, agents interact with their neighbors frequently and interchanging a small amounts of information (e.g., crowd evacuation, traffic or tissue growth simulations), but, in other cases, agents interactions can be between any pair of agents (e.g., social networks, or economy simulations) or interchange larger amounts of information (e.g., stock exchange simulation).

Consequently, the benchmark must be able to replicate the most usual interaction patterns among agents and allow to configure these interactions (pattern, frequency, size) in order to determine their influence in the communication and synchronization overhead.

In the second place, besides interactions among them, another relevant factor affecting performance is the computation performed by the agents. This computation is always expressed in the form of functions or methods in the considered frameworks. The overall ABMS performance can be influenced by the global workload, its distribution and its ratio against communication/synchronization.

Usually, the amount of computation performed by an agent between consecutive interactions is relatively small, for example, computing the agent's next position in accordance with its neighbors position or exerted force. However, in other cases the amount of computation can be significant, such as in the case of complex sociological simulations involving many input factors.

Therefore, in order to evaluate its impact on the application's performance, the benchmark must be also able to represent different workloads, agent distribution and communication/computation ratio.

Finally, analyzing the scalability and efficiency of a parallel application is a significant element of its performance evaluation. Consequently, the benchmark must not impose limitations on the number of agents or resources that can be used in the evaluation, beyond of those imposed by the generation framework or the underlying hardware.

3.2 Benchmark design

In accordance with the requirements established in the previous section, the user must be able to configure the following simulation aspects:

1. **Communication volume, frequency and pattern** For evaluating the impact of these factors, the benchmark includes the following parameters of the agents interactions:
   - Number of extra bytes involved in each interaction (*num_bytes*).
   - Number of agents that will be involved in each interaction (*num_agents*).
   - The maximum distance between interacting agents (*distance*).
2. **Amount of computation** For controlling the amount of computation performed by the simulator, the benchmark includes a parameter to set the amount of extra work done by each agent in each iteration (*table_sz*).
3. **Distribution and evolution of the workload** In order to control the workload distribution, the benchmark includes, two parameters which indicate the probability of generating and eliminating agents, respectively, and the location in the space where these probabilities hold (these probabilities decrease linearly as the agents move away from these locations).

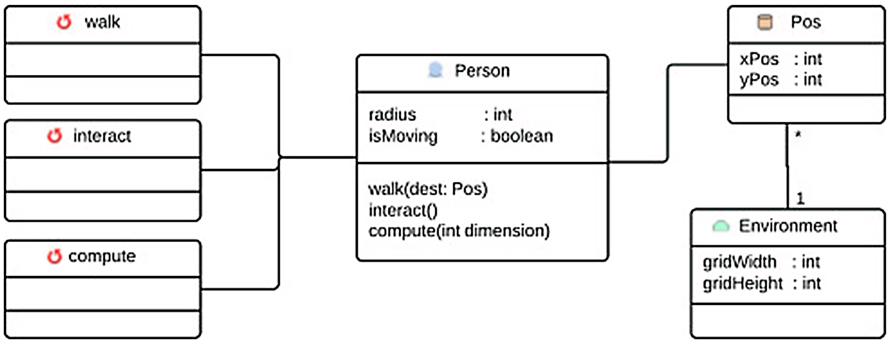


Fig. 3 AML representation of the base agent model [11]

4. **Size of the test** For controlling the size of each test, the initial number of agents, the size of the environment, number of iterations of the simulation and the number of computing elements are parameters of the benchmark.

The benchmark design is based in the adaptation done in [11] of the *prisoner's dilemma* implementation used in a Repast HPC demo. Figure 3 shows the agent model language (AML) representation of this base agent model.

The agent behavior is divided in the three following states, executed in each simulation iteration:

1. The *walk state* allows agents to move in a random direction on the environment, to one of its 8 Moore neighbor cells in the grid.
2. The *interact state* allows agents to interact with *num_agents* agents in their perception field (determined by *distance*). The message is composed of the agent id and an integer value plus a *num_bytes* number of arbitrary bytes. This state simulates communications between agents for assessing the communication impact on each platform.
3. The *compute state* simulates the workload generated by running the agent inner algorithms. This state computes a one-dimensional forward discrete Fourier transform (DFT) [27] on a table of size *table_sz*.

We are aware that a complete formal specification of the benchmark model and parameters must be provided in order to assure its fairness and effectiveness. In addition, a clear set of validation checks must be defined to ensure that the benchmark has been properly implemented. However, the goal of this work is to propose a general benchmark analysis and design, providing enough evidence to support our hypothesis and decisions, and to present and discuss it with the ABMS community. Next, if a consensus is reached, it will be worth taking the effort to work on the benchmark detailed specification and validation.

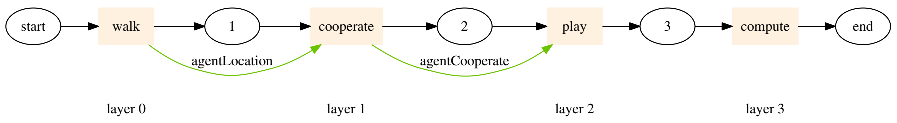


Fig. 4 State diagram of the agent model implemented in FLAME

3.3 Benchmark initial implementation

This section introduces the main aspects of an initial implementation of the benchmark using FLAME, FLAME GPU, EcoLab and Repast HPC, some of the most well-known parallel ABMS generation platforms.[1]

3.3.1 FLAME

Figure 4 shows the state diagram of an agent in the benchmark implementation using FLAME. The behavior of every agent in each simulation iteration can be described in the following way accordingly to this state diagram:

1. *walk* Move randomly to one of its 8 Moore neighbor cells and send a message (*agentLocation*) to the *board* indicating its new position.
2. *cooperate* Read all the messages on the board and determine *num_agents* agents within its perception field (defined by *distance*). Send a message (*agentCooperate*) to the board for each agent in the perception field. Each message includes the source agent id (the one looking for cooperation), the destination agent id (the one in its perception field), the cooperation decision (1 = cooperate, 0 = do not cooperate) and *num_bytes* extra bytes.
3. *play* Using an input message filter, process every message sent to the agent and compute the payoff according to the its cooperation decision and the one of the source agents.
4. *compute* Compute a one-dimensional DFT on a table of size *table_sz*. Moreover, in this state the parameters for controlling the workload evolution are taken into consideration. The agent computes the probability of creating a new agent (*Pb*) and the probability of dying (*Pd*) according to its distance to the positions with the maximum probability for creating a new agent or dying, respectively. Both probabilities decrease linearly with the distance to the maximum probability positions. Next, a new agent is created with a *Pb* probability and the agent dies with a *Pd* probability.

The diagram in Fig. 4 has been extracted directly from the XML specification of the agents, and each transition described in the behavior has been programmed in C.

[1] The implementations for FLAME, FLAMEGPU and Repast HPC are, respectively, available in https://github.com/HPCA4SE-UAB/ABMS-Benchmark-FLAME, https://github.com/HPCA4SE-UAB/ABMS-Benchmark-FLAMEGPU, https://github.com/HPCA4SE-UAB/ABMS-Benchmark-Repast.

3.3.2 FLAME GPU

The FLAME GPU model has the same structure as the FLAME model. The evolution of the model is represented by the following functions:

1. *walk* Move randomly to one of its 8 Moore neighbor cells and create a message (*agentLocation*) indicating its new position.
2. *cooperate* Read all messages (*agentLocation*) and determine *num_agents* agents within its perception field (defined by *distance*). Send a message (*agentCooperate*) for each agent in the perception field. Each message includes the source agent id (the one looking for cooperation), the destination agent id (the one in its perception field), the cooperation decision (1 = cooperate, 0 = do not cooperate) and *num_bytes* extra bytes. This has been implemented in a different way than in FLAME because a restriction of FLAME GPU. This restriction is that FLAME GPU does not support data arrays and any extra byte needed on the messages has to be implemented with simple type variables.
3. *play* Using an input message filter, process every message sent to the agent and compute the payoff according to the its cooperation decision and the one of the source agents.
4. *compute* Compute a one-dimensional DFT on a table of the same size than in the FLAME benchmark version, *table_sz*. FLAME computes this DFT with the libfftw3 library which does not exist in CUDA. In its place, CUDA has cuFFT library, the NVIDIA CUDA fast Fourier transform product [28]. We have combined the cuFFT with the CUDA Streams [29] to improve performance. Given that cuFFT library works as a CUDA kernel, before doing the DFT implementation we have done some tests to be able to see what number of CUDA Streams offers better performance using cuFFT for DFT calculation. Finally, we decided to use three CUDA Streams combined with cuFFT library because this number of CUDA Streams shows better performance when simulating a significant number of agents.

The first three functions have been directly translated from the FLAME model taking into account the FLAME GPU particularities, while the *compute* function has been fully restructured according to the requirements of the cuFFT CUDA library.

3.3.3 Repast HPC

The initial Repast HPC implementation of the benchmark includes two classes: *Model* and *Agent*.

The *Model* class:

- Creates the environment (context in Repast terminology) using a discrete space projection for defining the grid where the agents interact,
- Implements the methods for packing and unpacking agents, needed for replicating and migrating agents in/to other simulation processes.
- Creates the agents assigned to each simulation process.
- Sets the events that will be executed by the simulator scheduler.

Listing 1 Model method implementing agents behavior and synchronization among simulation processes in Repast HPC

```
1  void Model::doSomething(){
2          int whichRank = 0;
3
4          vector<Agent*> agents;
5          context.selectAgents(..., countOfAgents, agents);
6          vector<Agent*>::iterator it = agents.begin();
7          while(it != agents.end()){
8                  (*it)->play(&context, discreteSpace);
9                  (*it)->compute();
10                 it++;
11         }
12   it = agents.begin();
13   while(it != agents.end()){
14          (*it)->move(discreteSpace);
15          it++;
16   }
17   discreteSpace->balance();
18   RepastProcess::instance()->synchronizeAgentStatus(
         context, ...);
19   RepastProcess::instance()->synchronizeProjectionInfo(
         context, ...);
20   RepastProcess::instance()->synchronizeAgentStates(...);
21 }
```

– Implements methods to be executed in each event. In particular, it implements the *doSomething* method shown in Listing 1 that is executed once in each simulation iteration. This method performs the *play–compute–walk* behavior for every agent assigned to a simulation process (lines 4–16), and also, calls the methods that automatically synchronize all simulation processes (lines 17–20). It is worth noticing that these are the methods that fire inter-process communications.

The *Agent* class includes the methods for moving the agent (*move*), interact with its neighbors (*play*) and *compute*. Most of the benchmark parameters are considered in this class, the *play* method takes into consideration the *distance* and *num_agents* parameters and the *compute* method uses the *table_sz* parameter. Finally, the *num_bytes* parameter is used to define a buffer in this class, which artificially increases the size of the instantiated objects (actual agents).

3.3.4 EcoLab

The initial implementation of the benchmark in EcoLab is composed of a set of classes, which hierarchy is shown in Fig. 5, and a Tcl script.

The *Agent* class includes the methods that implement the agent's behavior (*compute* and *cooperate*) and the variables that define its state. The benchmark parameters *table_sz* and *num_bytes* are used in this class to control the amount of computation of each agent (compute method) and the amount of bytes communicated (defining a buffer that artificially increases the agent's size). Moreover, Classdesc is used for

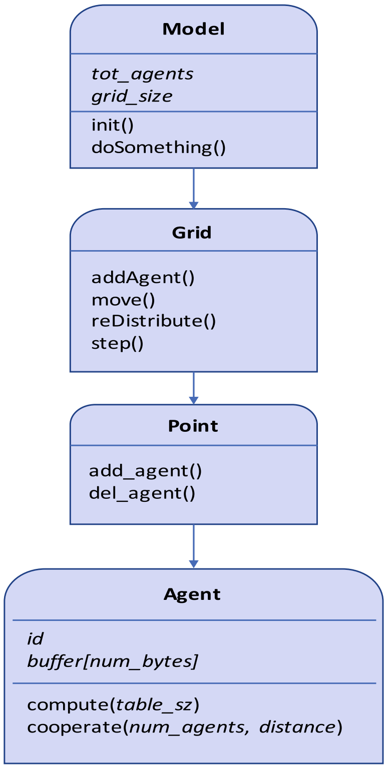


Fig. 5 Benchmark Class hierarchy in EcoLab

automatically generating the serialization/deserialization functions for objects of this class.

The *Point* class that implements the graph nodes where the agents are placed. This class inherits from the Graphcode *object* class, and consequently, EcoLab provides methods for migrating objects of this class.

The *Grid* class represents the graph where the agents are placed (similar to the Repast HPC context). It allows the Model to iterate through the graph and access the agents in each node (grid cell). For this reason, the method that implements the play–compute–walk behavior for every agent assigned to a simulation process belongs to this class (*step*). The benchmark parameters *num_agents* and *distance* are used in this class to determine the set of agents that will interact among them. Additionally, this class inherits from the Graphcode *graph* class, which includes methods for distributing and redistributing the graph among the simulation processes.

The *Model* class, which inherits from Grid, holds the initial model parameters and is used to initialize the environment and to create the agents. It includes a method to let the simulation go step by step (*doSomething*). In addition, using Classdesc, this class is declared to be accessible from the Tcl script that controls the simulation process.

Finally, the Tcl script calls the initialization function included in the Model class, which creates the environment and the initial agent population, and then executes the specified number of simulation iterations. Consequently, the benchmark parameters that control the test size are considered in this script.

4 Benchmark evaluation

This section shows four different case studies using the implementations described in Sect. 3.3: first, results obtained from the comparison between FLAME and Repast HPC; next, the effect on performance of extending FLAME with an automatic and dynamic load balancing mechanism; then, the comparison between FLAME GPU and FLAME; and finally, the comparison between Repast HPC and EcoLab.

In all the experiments, 100 simulation steps have been executed at least five times. The results shown in each case correspond to the average of these executions.

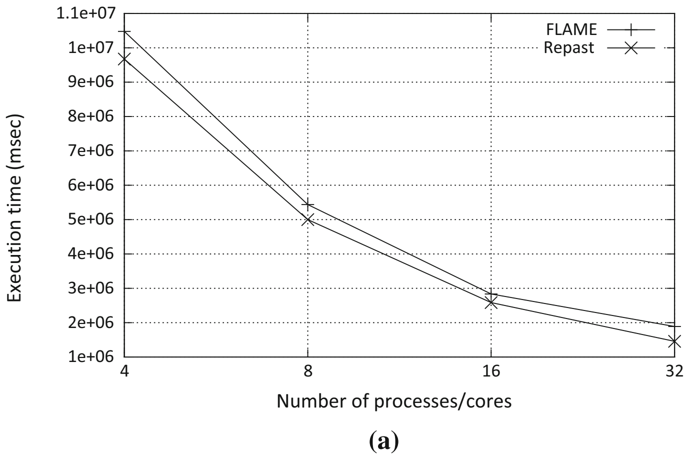
The experiments have been executed in different hardware platforms:

– Wilma: 12 nodes with 2 AMD Opteron 4180 processors (6 cores/processor, 128 KB of L1, 512 KB of L2 and 6 MB of L3 shared) and 8 GB of main memory.
– Batman: 2 nodes with an AMD Opteron 6376 processors (16 cores/processor, 768 KB of L1, 16 MB of L2 shared and 16 MB of L3 shared) and 256 GB of main memory.
– Gpt: one node with 2 Intel Haswell Xeon E5-2620 version 3 processors (6 cores/processor with Hyperthreading, 64 KB of L1, 256 KB of L2 and 15 MB of L3 shared) and 64 GB of main memory, and 2 Nvidia GPUs, one GTX 1080 (8 GB main memory and 2560 CUDA cores), and one GTX Titan Black (6 GB main memory and 2880 CUDA cores).

4.1 FLAME versus Repast HPC

First, we have conducted a set of experiments on Batman in order to evaluate the scalability of the simulators generated by these frameworks. For the strong scalability test, the number of processes and cores has been variated from 4 to 32, using a fixed number of 10,000 agents (*num_agents*), an extra size of 256 B for each interaction (*num_bytes*), a *distance* of 10 units in world of 300 $\times$ 300 units and an extra amount of work for each agent corresponding to the computation of a FFT on a table of 16 KB (*table_sz*). Figure 6a shows that Repast HPC presents a slightly better scalability than FLAME (speedup of 6.64 vs. 5.54, respectively, when increasing the number of cores from 4 to 32), which is likely caused by the significantly lower (2 orders of magnitude) communication load shown in Fig. 6b.

For the weak scalability test, the agents' parameters are kept the same (*distance*, *num_bytes* and *table_sz*), the number of processes/cores has been fixed to 32, and the number of agents has been variated from 2000 to 10,000 (*num_agents*). Figure 7a shows that for Repast HPC a fivefold increase in the workload leads to a 4.7$\times$ increase in the execution time (linear increase), while for FLAME it leads to a 38.6$\times$ increase (quadratic increase). Again, the communication overhead seems to explain this difference. Figure 7b illustrates that the communication load is 2 orders of magnitude higher for FLAME and that it grows faster than the one for Repast. However, FLAME is sig-



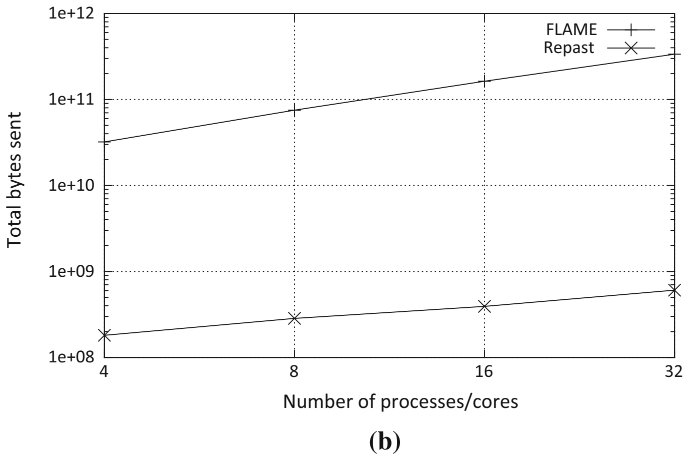
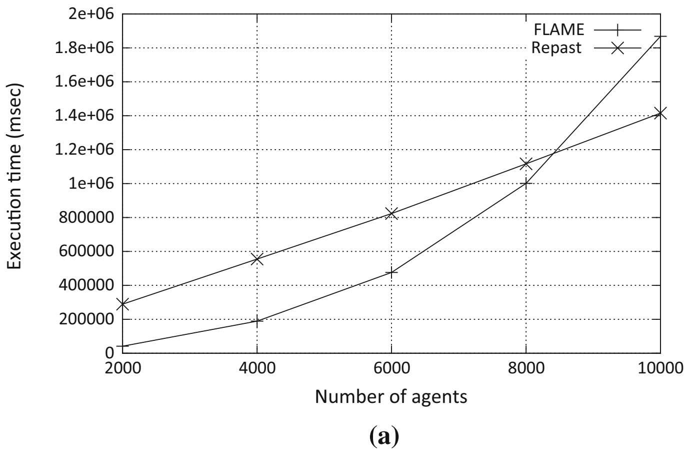


Fig. 6 FLAME versus Repast HPC: scalability results from 4 to 32 cores. **a** Execution time, **b** total bytes sent

nificantly more efficient doing the computation (approximately 10×), which explains its better results for smaller workloads.

Next, we have conducted two experiments in order to study the influence of communication in both platforms. In the first experiment, executed on Batman, all the parameters have been fixed (10,000 agents, 32 processes/cores, 16 for *table_sz* and a *distance* of 10) except for *num_bytes*, which has been assigned 16, 32 and 256 B. Figure 8a shows how the increase in the size of agent interactions is sharply affecting FLAME's execution time (2.5× increase), while barely affecting Repast HPC (only an increment of 10%). The reason can again be explained by results in Fig. 8b, FLAME is communicating a least 100 times more bytes than Repast HPC, and in addition, increasing the size of agents interactions has a bigger effect in FLAME. (From 16 to



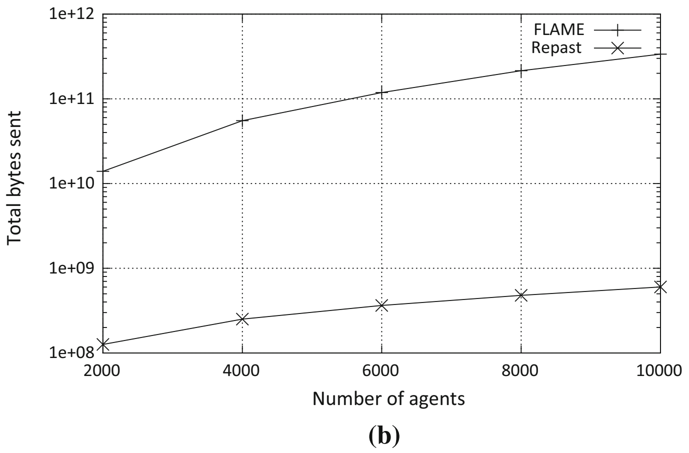


Fig. 7 FLAME versus Repast HPC: scalability results from 2000 to 10,000 agents. **a** Execution time, **b** total bytes sent

256 B, in FLAME overall communication grows 10×, while in Repast HPC one only grows 4×.)

The second experiment, executed on Wilma, tries to stress Repast HPC communication approach. FLAME's performance problem with communication is a consequence of its centralized communication library (*libmboard*). However, this approach has the advantage of allowing to communicate any pair of agents disregarding their placement. On the contrary, Repast HPC is handling communication efficiently, but communicating any arbitrary pair of agents is not as straight forward. One way for increasing the number of potential communicating agents in Repast HPC consists in increasing the size of the overlapping buffer between neighbor simulation processes, which is related to the *distance* parameter of the benchmark. Therefore, we have executed the simulation setting *distance* to 50, 100, 150, 200 and 250 units, which corresponds

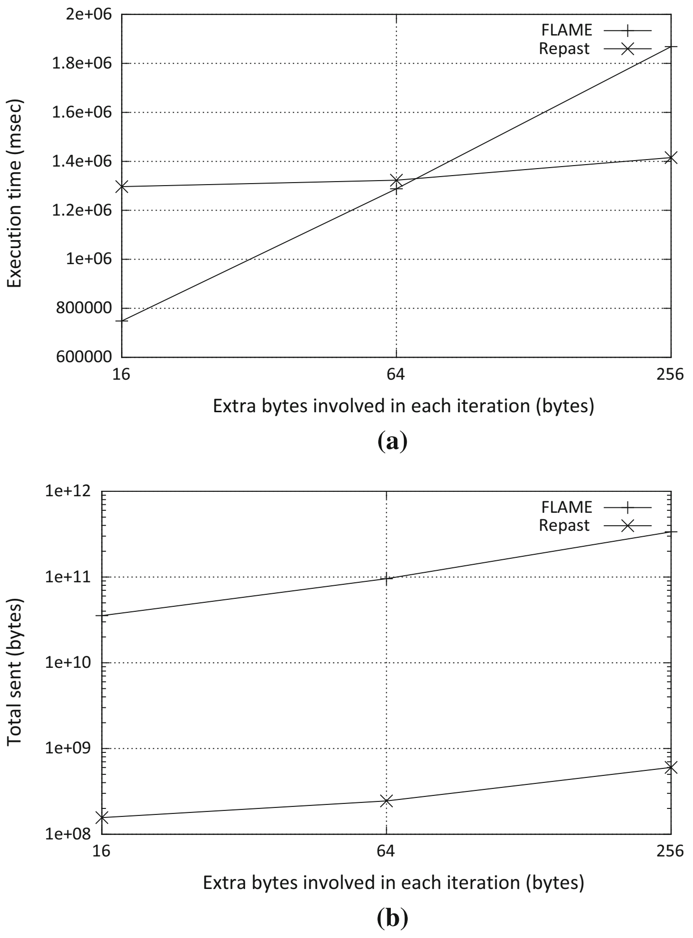
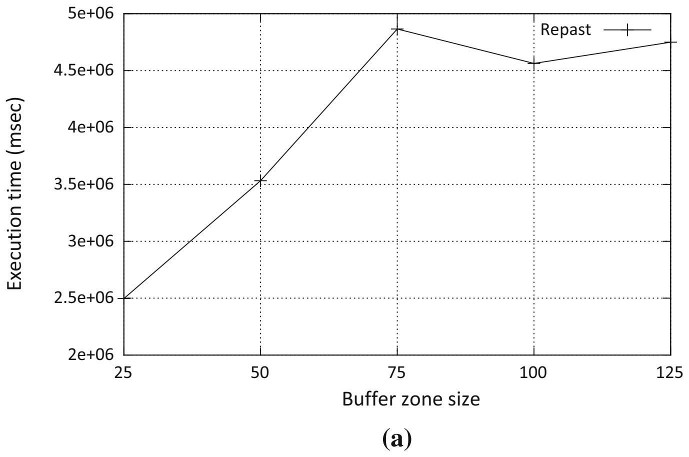


Fig. 8 FLAME versus Repast HPC: results variating the extra communication load (*num_bytes*) from 16 to 256 B. **a** Execution time, **b** total bytes sent

to overlapping buffers of 25, 50, 75, 100 and 125 units, respectively. The number of processes/cores has been set to 16 and the other parameters have not changed.

Figure 9 shows that execution time and total sent bytes increase sharply when increasing the overlapping buffer, although the number of bytes sent is still significantly smaller than the one for FLAME. In addition, the figure also shows that the overlapping buffer only affects processes that are direct neighbors because there is no penalty beyond a size of 75 units, which is the area that is assigned to each process. (An area of 300 $\times$ 300 units divided among 16 processes leads to an area 75 $\times$ 75 units/process.) Consequently, we are not actually achieving the possibility of communicating any pair of agents.

In general, Repast HPC shows a better performance than FLAME because of its lower communication overhead. FLAME uses a centralized communication mechanism that allows communication of any pair of agents a higher cost. On the other hand,



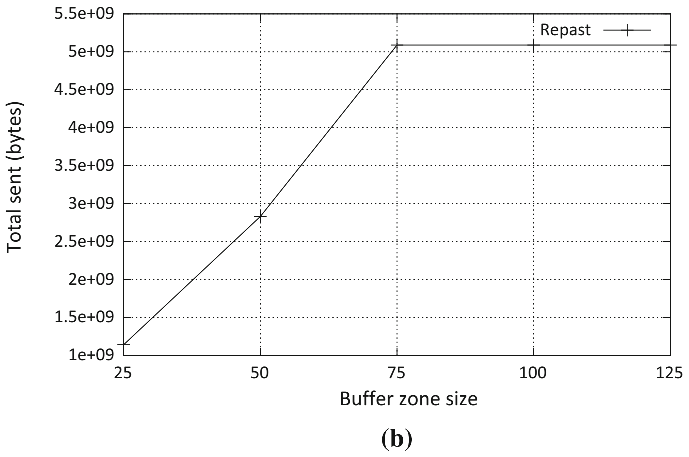


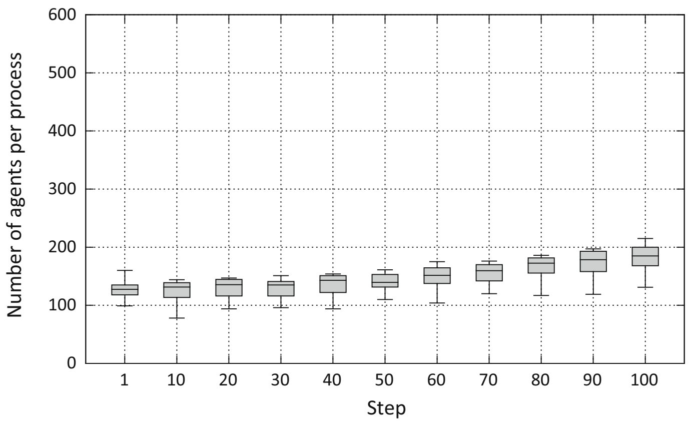
Fig. 9 FLAME versus Repast HPC: results variating the buffer zone size from 25 to 125 units. **a** Execution time, **b** total bytes sent

Repast HPC has a communication mechanism based on the replication of agents in different nodes, which is significantly more difficult to program, but which efficiently manages communication.

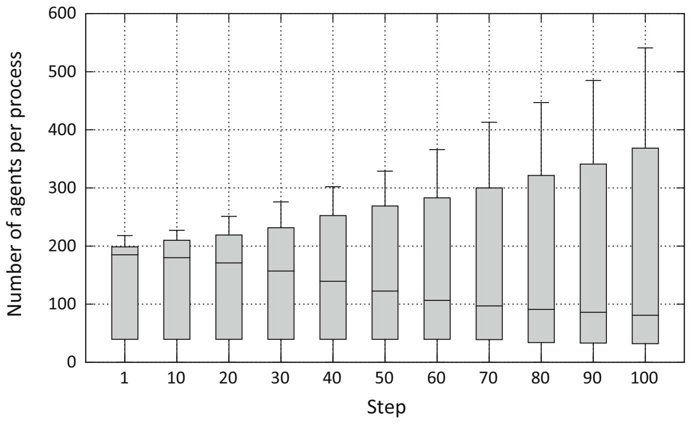
4.2 Extended FLAME versus FLAME

This section presents a set of experiments, executed on Wilma, for assessing the impact of the mechanism for automatic and dynamic load balancing incorporated in the FLAME extension described in Sect. 2.1.1.

The number of processes and cores has been fixed to 32, the number of agents to 4000 (*num_agents*), an extra size of 256 B for each interaction (*num_bytes*) and an extra amount of work for each agent corresponding to the computation of a FFT on a



(a)



(b)

Fig. 10 Extended FLAME versus FLAME: evolution of the number of agents per process. **a** Extended FLAME, **b** FLAME

table of 16 KB (*table_sz*). In addition, we have added two parameters which indicate (1) the maximum probability of generating and eliminating agents, set to 2%, and (2) the location in the space where these probabilities hold, set to (50, 50) for elimination and (150, 150) for generation. These probabilities decrease linearly as the agents move away from these locations.

Figure 10 shows the statistical distribution, using boxplots, of the number of agents per process every 10 simulation steps. It clearly demonstrates the positive effect of the automatic and dynamic load balancing mechanism of Extended FLAME because the number of agents is more balanced among processes in this case. This better load balancing is noticed in the simulation execution time, which decreases from 1199.312 s in FLAME to 981.312 s in Extended FLAME, reaching a global improvement of 18.2%.

4.3 FLAME GPU versus FLAME

This section presents a set of experiments, executed on Gpt, to evaluate the performance of the benchmark generated by FLAME GPU (massive parallel processing on GPUs) using different message partition strategies, different communication loads and different devices. In addition, we compare the results obtained by FLAME GPU with the ones generated by FLAME (distributed parallelism using MPI).

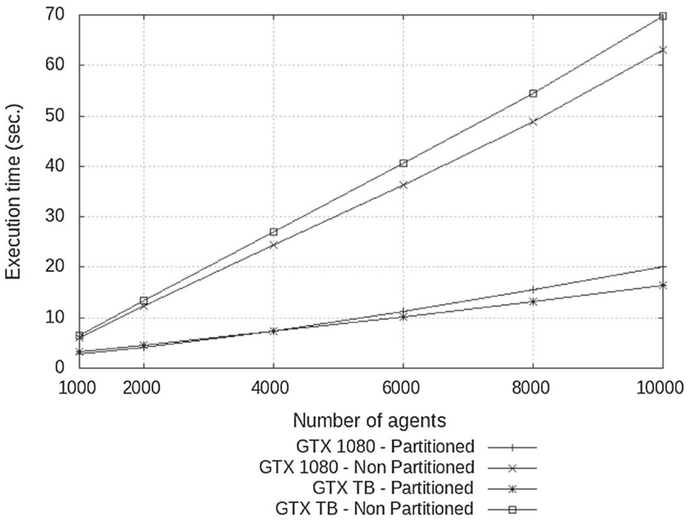
For these experiments, we have used an agent number range from 1000 to 10,000 agents (*num_agents*), with or without an extra communication load of 64 B for each interaction (*num_bytes*), a *distance* of 10 units in a world of 300 $\times$ 300 units and an extra amount of work for each agent corresponding to the computation of a FFT on a table of 16 KB (*table_sz*).

In the first set of experiments, executed on both available GPUs, we have used the benchmark for evaluating the effect of message partitioning for different communication loads. The test consists in running the benchmark using *non-partitioned* and *partitioned* messages and with or without an extra load of 64 B on each message. The size of the original message is 32 B because it only contains the source and destination agent ids and the cooperation decision.

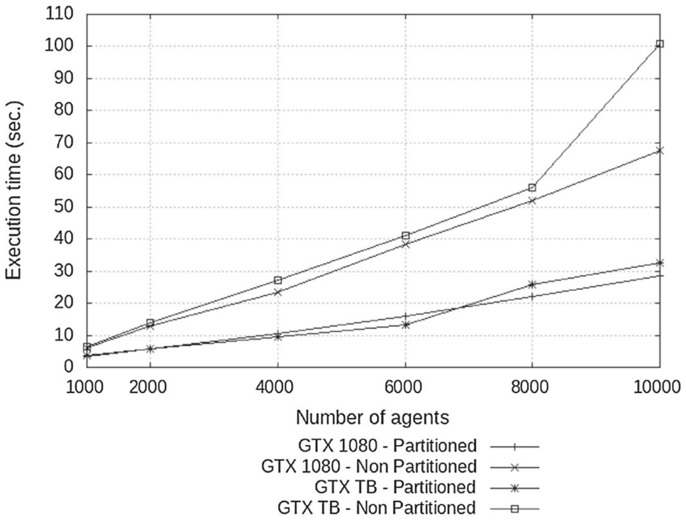
Figure 11a shows, in the first place, the positive impact of using spatial partitioned space to process messages without introducing additional communication load. The difference between using non-partitioned and partitioned messages is smaller (2.07$\times$ or 3$\times$) when the simulation is executed with a small number of agents, but it increases (up to 4.24$\times$) as the number of agents increases. In addition, it can be seen that for non-partitioned space the GTX 1080 GPU is consistently outperforming the Titan Black GPU. This can be explained by the faster clock rate of the GTX 1080 GPU for computation and memory access. However, in the partitioned space case it is the other way around from above 4000 agents. Finally, these figures show that in all cases a linear increase in the workload causes a linear increase in the execution time. In the case of non-partitioned messages, increasing the work load 10 times produces a 10.5$\times$ increase in the execution time, while for partitioned space, the same workload increase causes a 5.2$\times$ and 6.9$\times$ execution time increase for the Titan Black GPU and the GTX 1080 GPU, respectively.

Figure 11b shows, in the first place, the advantage of using spatial partitioned space to process messages which size has been increased by 64 B. The impact of increasing the size of agent interactions is negligible in the executions using the GTX 1080 GPU. However, a significant performance degradation appears when executing the benchmark on the Titan Black GPU for higher number of agents. This may be explained by the fact that the Titan Black GPU has 15 SM with 192 CUDA cores each, while the GTX 1080 GPU has 20 SM with 128 CUDA cores each. As we increase the number of agents, the message structure is bigger and this can degrade global and share memory efficiency. Nevertheless, new tests increasing the number of agents and the communication load must be done to be able to confirm this hypothesis.

Finally, we have compared the execution time obtained for the worst case in FLAME GPU (non-partitioned messages and extra communication load of 64 B) with the results obtained with FLAME on Batman using 32 simulation processes. Figure 12a shows



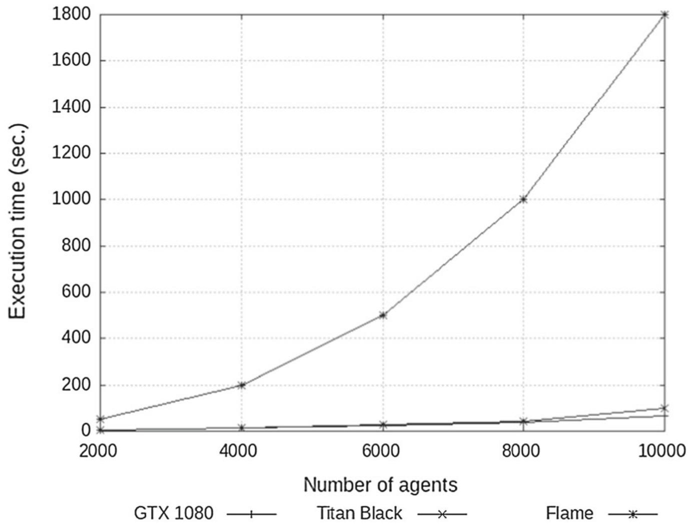
(a)



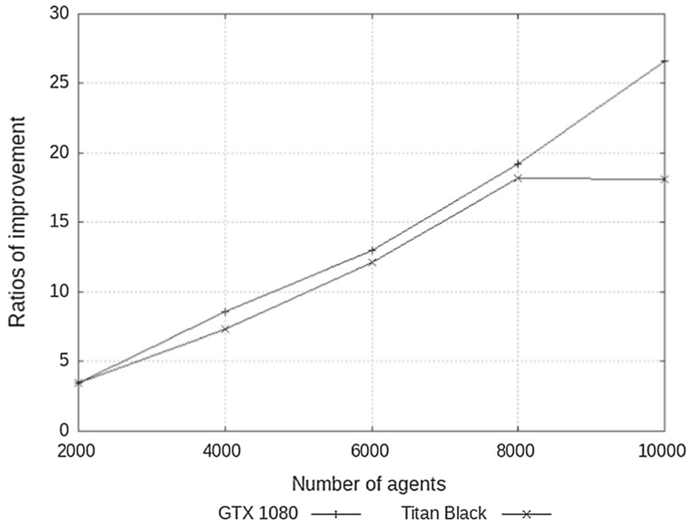
(b)

Fig. 11 FLAME GPU versus FLAME: scalability results from 1000 to 10,000 agents. **a** Execution time without extra communication load, **b** execution time with extra communication load

that the performance improvement is significantly higher as the number of agents increases, i.e., for bigger workloads the GPU version of the simulation is giving much better results than the multi-CPU version. Figure 12b shows the ratio of improvement for both GPUs, GTX 1080 and Titan Black, with respect to the FLAME version. For example, running the simulators for 10,000 agents, an extra communication load



(a)



(b)

Fig. 12 FLAME GPU versus FLAME: execution time comparison. **a** Execution time. **b** Improvement ratio

of 64 B, FLAME GPU obtains 70 seconds with GTX 1080 GPU and 100 seconds for Titan Black GPU, while FLAME reaches 2000 seconds with 32 processes; that is, Titan Black GPU and and GTX 1080 GPU deliver 20× and 25.8× performance improvement, respectively. These results corroborate the claim made by FLAME GPU authors about the performance advantages when using simulators generated for GPUs (see Sect. 2.2).

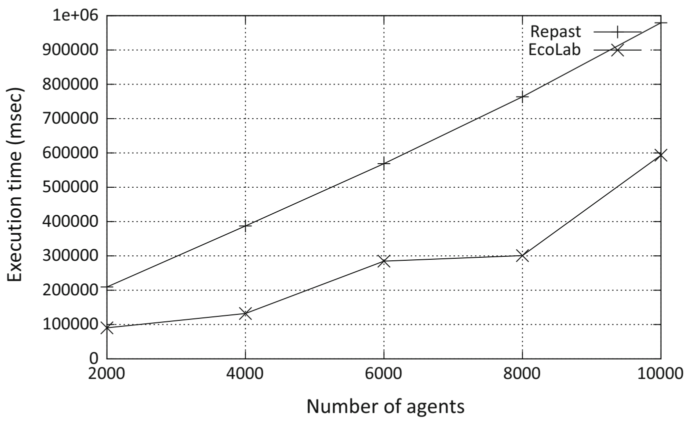


Fig. 13 EcoLab versus Repast HPC: scalability results from 2000 to 10,000 agents

4.4 EcoLab versus Repast HPC

The EcoLab version of the benchmark has been the last one to be developed, and hence, it is still necessary to conduct more validation checks to be sure that it complies with the specification. However, the preliminary tests show promising results.

For example, Fig. 13 illustrates the results of executing the benchmark implemented in EcoLab in comparison with the results obtained with Repast HPC. In this test, we evaluate the weak scalability applying the configuration described in Sect. 4.1, namely running on 32 cores in Wilma, an extra size of 256 B for each interaction (*num_bytes*), an extra amount of work for each agent corresponding to the computation of a FFT on a table of 16 KB (*table_sz*), a *distance* of 10 units and variating the number of agents from 2000 to 10,000 (*num_agents*). It can be seen that, for the parameter values used in this test, performance of the benchmark implemented in EcoLab is at least 2 times better than the one implemented in Repast HPC.

5 Related work

We have already mentioned several comparative studies [10,11] or [12], which, in some cases, use a specific model for testing the considered platforms. This approach has two main problems: On the one hand, the implemented model usually tests only some of the features of the compared platforms, and in the second place, the implementation could have some errors that may influence the results. This is the case for example of [11], which uses a modified version of one of the Repast HPC tutorials for evaluation purposes.

To the best of our knowledge, the most serious attempts to create a benchmark for assessing parallel ABMS applications performance comes from the OpenAB group [13] and were published in [30,31].

The first proposal, based in one of the FLAME demos, has been implemented for FLAME GPU [4], MASON [7] and REPAST Simphony [5]. Although this is a very nicely formalized proposal, it is limited to *fixed radius near neighbor* ABMS.

The second proposal, which consists in modeling a set of molecules following Brownian dynamics methods, has been implemented for FLAME GPU. This benchmark introduces the variation of the system workload during the execution, which has been also considered in our design. However, the model is still more restrictive than the one proposed in this work because it does not consider parameters such as the message size, the amount of computation or the number of agents that interact within a certain distance.

6 Conclusions and future work

In this work, we have proposed and designed a benchmark for assessing the performance of parallel ABMS applications. This benchmark takes into consideration the most common characteristics of this type of applications and includes parameters for influencing their relevant performance aspects. In this way, the user can control: (1) communication features, such as the communication pattern and message size and frequency, (2) application total workload and its distribution among simulating processes and (3) the size of each test.

We have also provided an initial implementation of the benchmark for FLAME, FLAME GPU, Repast HPC and EcoLab, and used them for comparing the performance of the simulators generated by these platforms. The obtained results are mostly in agreement with previous studies, but the designed and implemented specification has allowed for testing a wider set of aspects and obtaining more reliable results.

We expect to share and discuss this proposal with the OpenAB community to enhance it, produce a detailed specification of the benchmark, and develop new and improved versions for the most relevant parallel ABMS generation platforms.

References

1. Macal CM, North MJ (2009) Agent-based modeling and simulation. In: Proceedings of the 2009 Winter Simulation Conference (WSC), pp 86–98
2. National Research Council (2014) Advancing land change modeling: opportunities and research requirements. The National Academies Press, ISBN 978-0-309-28833-0
3. Coakley S, Gheorghe M, Holcombe M, Chin S, Worth D, Greenough C (2012) Exploitation of high performance computing in the FLAME agent-based simulation framework, high performance computing and communication 2012. In: IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp 583–545
4. Richmond P, Coakley S, Romano DM (2009) A high performance agent based modelling framework on graphics card hardware with CUDA. In: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09), vol 2, pp 1125–1126
5. Collier N, North M (2013) Parallel agent-based simulation with repast for high performance computing. Simulation 89(10):1215–1235
6. Standish RK (2008) Going stupid with EcoLab. Simulation 84(12):611–618

7. Cordasco G, De Chiara R, Mancuso A, Mazzeo D, Scarano V, Spagnuolo C (2013) Bringing together efficiency and effectiveness in distributed simulations: the experience with D-MASON. Simulation 89(10):1236–1253
8. Rubio-Campillo X (2014) Pandora: a versatile agent-based modelling platform for social simulation.In: Proceedings of SIMUL 2014, The Sixth International Conference on Advances in System Simulation, PP 29–34
9. Borges F, Gutierrez-Milla A, Luque E, Suppi R (2017) Care HPS: a high performance simulation tool for parallel and distributed agent-based modeling. Future Gener Comput Syst 68:59–73
10. Railsback SF, Lytinen SL, Jackson SK (2006) Agent-based simulation platforms: review and development recommendations. Simulation 82(9):609–623
11. Rousset A, Herrmann B, Lang C, Philippe L (2016) A survey on parallel and distributed multi-agent systems for high performance computing simulations. Comput Sci Rev 22:27–46
12. Abar S, Theodoropoulos GK, Lemarinier P, OHare GMP (2017) Agent based modelling and simulation tools: a review of the state-of-art software. Comput Sci Rev 24:13–33
13. Open Agent Benchmark Initiative for Parallel and Distributed Benchmarking. www.openab.org. Accessed 16 Nov 2018
14. Márquez C, César E, Sorribes J (2013) Agent migration in HPC systems using FLAME, Euro-Par 2013: Parallel Processing Workshops, pp 523–532
15. Márquez C, César E, Sorribes J (2015) Graph-based automatic dynamic load balancing for HPC agent-based simulations, Euro-Par 2015: Parallel Processing Workshops—Euro-Par 2015 International Workshops, pp 405–416
16. Kang G, Márquez C, Barat A, Byrne AT, Prehn JHM, Sorribes J, César E (2017) Colorectal tumour simulation using agent based modelling and high performance computing. Future Gener Comput Syst 67:397–408
17. Devine K, Boman E, Heaphy R, Bisseling R, Catalyurek U (2006) Parallel hypergraph partitioning for scientific computing. In: 20th International Parallel and Distributed Processing Symposium, IPDPS 2006, p 10
18. Richmond P, Walker D, Coakley S, Romano D (2010) High performance cellular level agent-based simulation with FLAME for the GPU. Brief Bioinform 11(3):334–347
19. FLAMEGPU Documentation and User Guide. http://docs.flamegpu.com/en/latest/. Accessed 16 Nov 2018
20. The Apache XML project, Xalan-Java. https://xml.apache.org/xalan-j/. Accessed 16 Nov 2018
21. The OpenGL Extension Wrangler Library. http://glew.sourceforge.net/. Accessed 16 Nov 2018
22. CUDA Data Parallel Primitives Library. http://cudpp.github.io/. Accessed 16 Nov 2018
23. Tool Command Language. http://www.tcl.tk/. Accessed 16 Nov 2018
24. Standish RK, Madina D (2006) Classdesc and Graphcode: support for scientific programming in C++, CoRR. arXiv:cs/0610120. Accessed 16 Nov 2018
25. Karypis G (2011) METIS and ParMETIS, encyclopedia of parallel computing. Springer, Berlin, pp 1117–1124
26. Message Passing Interface (mpi-forum.org)
27. Frigo M, Johnson SG (2005) The design and implementation of FFTW3. Proc IEEE 93(2):216–231
28. The NVIDIA CUDA Fast Fourier Transform library. https://docs.nvidia.com/cuda/cufft/index.html. Accessed 16 Nov 2018
29. The NVIDIA CUDA Streams. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf. Accessed 16 Nov 2018
30. Chisholm R, Richmond P, Maddock S (2016) A standardised benchmark for assessing the performance of fixed radius near neighbour, Euro-Par 2016: Parallel Processing Workshops, pp 311–321
31. Alzahrani E, Richmond P, Simons AJH (2017) A formula-driven scalable benchmark model for ABM, applied to FLAME GPU, Euro-Par 2017: Parallel Processing Workshops, pp 703–714

Affiliations

Andreu Moreno[1,2] · Juan J. Rodríguez[3] · Daniel Beltrán[2] · Anna Sikora[2] · Josep Jorba[3] · Eduardo César[2]

✉ Eduardo César
eduardo.cesar@uab.cat

Andreu Moreno
amoreno@euss.cat

Juan J. Rodríguez
juanjose.rodriguezg@uab.cat

Daniel Beltrán
daniel.beltranm@e-campus.uab.cat

Anna Sikora
anna.sikora@uab.cat

Josep Jorba
jjorbae@uoc.edu

1 Escola Universitària Salesiana de Sarrià (EUSS), Passeig Sant Joan Bosco, 74, 08017 Barcelona, Spain

2 Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, Bellaterra, Spain

3 Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya, Barcelona, Spain