A Synchronized and Dynamic Distributed Graph structure to allow the native distribution of Multi-Agent System simulations

Paul Breugnot, Bénédicte Herrmann, Christophe Lang and Laurent Philippe
DISC of FEMTO-ST, UBFC
Besançon, France
firstname.name@univ-fcomte.fr

***Abstract*—Multi-Agent Systems (MAS) are naturally good candidates for large-scale parallel simulations. However, implementing MAS simulations for distributed memory architectures, such as High Performance Computing clusters, is still complex for non-experts. In this article we present the principle of a Dynamic Distributed Graph structure, that enables the native distribution of MAS simulations. Most of the distribution related issues such as dynamic load-balancing, time synchronization and data migration across processes can be completely automated and abstracted for the user, who can safely design distribution independent MAS models. The major interest of our contribution is the transparent management of concurrent read / write requests across distant processes, a significant feature not provided by surveyed platforms. We also present FPMAS, an open source C++ implementation of a Distributed Multi-Agent System Simulation platform based on the Distributed Graph structure.**

***Index Terms*—multi-agent systems, simulation, hpc, distributed graph**

I. Introduction

Multi-Agent Systems (MAS) are a promising approach to simulate complex systems that cannot easily be modeled using linear equation systems. This is notably the case when interactions between heterogeneous entities should be represented, such as in micro or macro economic [1], biodynamic [2], epidemiological [3], crowd evacuation [4] or smart grid energy management [5] models for example. The main purpose of MAS simulations is usually to observe emergent properties produced by the simulation of simple behaviors applied to a large number of agents.

As the complexity of models grows, the field of MAS simulation might benefit from the usage of High Performance Computing (HPC) resources to raise execution time and memory limits. Since MAS are compound of autonomous agents that are assumed to perform their task independently from others, MAS are natural candidates for parallelization. However, the requirements for dynamic and stochastic interactions between agents is a challenging issue when attempting to execute a MAS simulation on **distributed memory** structures. The distribution of MAS models is thus a challenging task that induces lots of specific issues and, in consequence, HPC usage in MAS simulation is still limited, especially considering the fact that MAS modelers are usually not HPC experts.

Several platforms have been designed to attempt to solve those issues [6]–[8], but none of them are completely satisfying since they limit the possible interactions between agents located on distinct computing nodes thus requiring the final user to deal with distribution issues, such as remote writing. Hence MAS models are often required to be adapted in order to be compliant with an execution on distributed computing resources using those platforms.

In order to facilitate the design of Parallel and Distributed MAS simulations by non-experts, we propose a `DistributedGraph` framework that automates and abstracts the distribution related issues, such as dynamic load-balancing, time synchronization and data migration across processes to the user. This article presents the following contributions:

1) A formally defined `DistributedGraph` structure to allow to transparently maintain the continuity of a graph distributed across a set of processes.
2) A software architecture and its C++ implementation, FPMAS, to allow the `DistributedGraph` synchronization according to different modes, and its application to distributed MAS simulation.

In Section II we present some related works that motivate the development of the `DistributedGraph` approach. Section III introduces the `DistributedGraph` structure, that allows a **native and implicit** distribution of graph-based models over distributed computing architectures. A C++ implementation of a Distributed MAS simulation platform based on the `DistributedGraph` structure is described in Section IV, and finally we demonstrate its usage using a SIR epidemiological model example in Section V. A brief performance analysis is also presented.

II. Related Works

MAS modeling is a research field on its own. We particularly consider simulated agents as autonomous and interacting entities, executed by time step, possibly on an environment. In this context, many simulation platforms, languages and standards have been designed and improved along the past decades. We can notably cite the Repast toolkit [9], Netlogo [10], Influence/Reaction models [11], the GAMA platform [12] or SARL [13], among many others.

These platforms are however not suitable for HPC, and so their use is limited in the case of large simulations. To

overcome these limitations designers of MAS models may use Parallel and Distributed MAS simulation (PDMAS).

A. Parallel and Distributed Multi-Agent System simulation

Distributing a MAS simulation consists in splitting the global execution into a set of *processes*, in order to improve execution time and/or to solve limited memory issues. Each *process* is typically responsible for a subset of agents to execute.

We consider existing platforms [14] that support such distribution of MAS simulations, considering three important aspects:

1) Qualitative efficiency of the distribution, in term of load-balancing.
2) Complexity for non-expert to implement MAS models.
3) Constraints applied to models, that might be altered to run properly in distributed environments.

Repast HPC [6] is probably the most famous MAS simulation platform adapted to High Performance Computing. It has been used to simulate several millions of agents using up to 32,768 processes [6]. Concrete models, such as "ChiSIM" [15], have also been successfully implemented.

In order to distribute a model across computing nodes, Repast HPC splits the simulated environment into a regular grid, which size is the number of used processes. Agents contained in each cell are then executed on the associated process. To ensure the continuity of the environment, agents *near cell borders*, according to agents perception range, are replicated on neighbor processes. The area corresponding to replicated agents is called the "Overlapping Zone" (OLZ). This concept raises several issues:

- Agents in the OLZ can only be updated at the end of each time step, importing data from their origin process. So, within a time step, the replicated agent data cannot be guaranteed to be up to date, since the original data might be modified from the origin process within the time step.
- Any modifications applied to a replicated agent are **lost** at the end of the time step.

It is hence not possible to perform *write* operations across processes. Repast HPC still provides some utilities to *migrate* agents to other processes to allow agents on the target process to modify their data, but this implies to design many distribution related tricks, altering the original model.

FLAME and FLAME GPU [16] are also works of interest that use a different approach. Agents can only interact using a "message board": messages are exchanged at the end of each time step to trigger some agent behaviors. A relevant aspect of the project is the abstraction of low-level parallel features to the user using an XML based formalism, called X-Machines, to represent agents. However, explicit messages usage still limits agent interactions across CPU or GPU processes.

Finally, D-MASON [8] is another platform used to distribute MAS simulations. It provides interesting "framework level" features that allow a high degree of abstraction of parallel issues to the user. Distribution and load-balancing is ensured by a grid based decomposition of the environment. Remote interactions are based on the "Area of interest" (AOI) concept, that is very close to RepastHPC's OLZ. D-MASON prevents errors generated by write requests in the AOI by completely forbidding write operations between agents: each agent updates its own internal state at step i only using neighbors data of step $i-1$, what might alter or even prevent the simulation of some models.

RepastHPC and D-Mason are respectively based on the existing Repast and MASON platforms, to facilitate the distributed implementation of models for people already familiar with the sequential platforms. However they strongly rely on *spatial* MAS to automatically distribute MAS models, while spatial MAS are only a subset of MAS models. Moreover the grid based distribution is quite limited in terms of load-balancing. If the distribution of agents on the environment is not uniform, performances can be significantly altered, since the workload is not shared equitably among processes. This can be worsened if agents are expected to move during the simulation. As illustrated in [15], communication volumes across processes is also a critical aspect of model distribution, that is usually not efficiently minimized using grid based distributions [17].

B. Graph Partitioning

Balancing the workload while minimizing communication volumes can be achieved using graph and hypergraph partitioning algorithms [18], already implemented in C/C++ libraries such as Zoltan [19]. Dynamic load-balancing is even handled relatively easily using this kind of partitioning.

Moreover, graph usage might be extended to more generic problems, since any computation can easily be represented with graphs [20]. We also believe that graphs can represent a wider range of MAS models than grid based structures, considering that dependencies between agents can easily be represented as edges in a graph. Even grid based environments can be represented as graphs, considering neighboring relations as dependencies between regions. In this context, this work generalizes the "nested-graph" approach developed in [17] with the adaptable concept of `DistributedGraph` for a more generic PDMAS support.

III. The Dynamic Distributed Graph Structure

Some computation can conveniently be distributed on a set of processes when it is represented as a graph, using existing partitioning algorithms. Moreover, MAS models can conveniently be represented as graphs, with edges representing dependencies and communications between agents. Existing graph partitioning libraries however focus on the computation of partitions and load-balancing algorithms. Zoltan still provides utility functions to migrate nodes according to the computed partitions, but do not provide dynamic data continuity utilities or features to bound tasks to nodes and allow them to communicate or update their data. The `DistributedGraph` structure, than can be partitioned using those existing

softwares, introduces extra features to support data continuity and dynamic graph management.

In this section we define the theoretical `DistributedGraph` data structure and its properties, that allows to preserve data continuity across processes independently of the current partitioning of the graph. Properties demonstrated in this section constitute the base of FPMAS, since they prove that models implemented in FPMAS behaves **independently of the current graph distribution**.

A. Definitions

The following definitions are provided to show the generic usage of the `DistributedGraph` and how it can interface with existing graph studies. They are also used to demonstrate the mathematical robustness of the structure, especially considering the properties introduced in the next section.

Definition 1. *A graph is defined by G such as:*

- $G = (V, E)$
- $V = \{v_i, i \in [1, n]\}$ *a set of vertices*
- $E \subset V \times V$ *a* multiset *of directed edges*

Notice that multiple edges of E are allowed to link the same two vertices on different *layers*, a feature that proves to be useful to efficiently represent different types of interactions between vertices.

We also introduce the notation $(\overline{u, v}) \iff (u, v)$ *or* (v, u). So $(\overline{u, v}) \in E$ actually means $(u, v) \in E$ *or* $(v, u) \in E$.

Definition 2. *The neighborhood of a vertex v in the graph $G = (V, E)$ is defined as $\{v_i \in V, \exists (\overline{v, v_i}) \in E\}$.*

Note that in our context the neighbors set is built from incoming **and** outgoing edges, since each vertex can access them indifferently.

Definition 3. *A* partition *of the graph G is defined by P_N such as:*

- $P_N = \{U_i, i \in [1, N], U_i \subset V\}$
- $\bigcup_{i=1}^{N} U_i = V$
- $U_i \cap U_j = \emptyset, \forall i, j \in [1, N], i \neq j$

In practice, N will be the number of processes on which G should be distributed and U_i is the subset of vertices assigned to process i.

Definition 4. *The `DistributedGraph` DG_N associated to a partition P_N of G is defined by $DG_N = \{G_i, i \in [1, N]\}$ such that, $\forall i \in [1, N]$:*

- $G_i = (V_i, E_i)$
- $V_i = U_i \cup U'_i$
- $U'_i = \{v_k \in V, v_k \notin U_i, \exists v_l \in U_i, (\overline{v_k, v_l}) \in E\}$
- $E_i = \{(v_k, v_l) \in E, v_k \in V_i \wedge v_l \in V_i\}$

The global `DistributedGraph` DG_N is divided into N subgraphs G_i. Each subgraph is associated to a process, and is called the *local representation of DG_N on the process i*. Each local graph contains all the vertices of the subset U_i of P_N. Those vertices are called the `LOCAL` vertices of the process i. We also trivially add `LOCAL` edges connecting two `LOCAL` vertices to the subgraph G_i. Finally, we need to add the set U'_i of `DISTANT` vertices to G_i: those vertices correspond to vertices that are connected to at least one `LOCAL` vertex of U_i, but that are associated to another subset U_j of P_N. Such a vertex v_k is also called *the local representation of v_k on process i*. Finally, all `DISTANT` edges between a `LOCAL` vertex of U_i and a `DISTANT` vertex of U'_i are added to G_i.

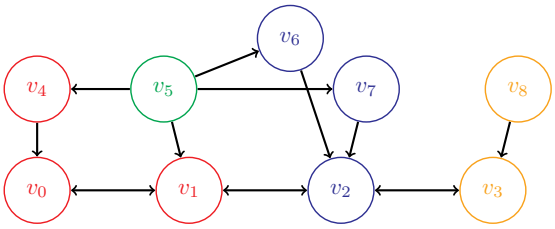
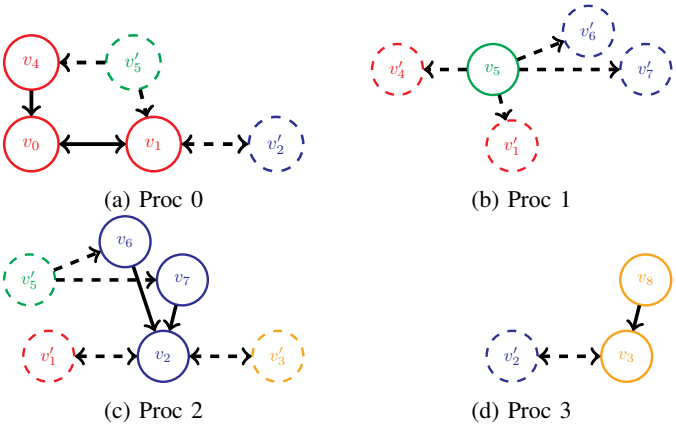


Fig. 1a: Initial Graph example. Each color corresponds to a subset of P_4



(a) Proc 0 (b) Proc 1

(c) Proc 2 (d) Proc 3

Fig. 1b: Graph Distribution example on 4 processes

On the example provided in figure 1, the initial graph represents a graph G with an initial partition P_4, with subsets U_i represented as different colors. The graph is then distributed on 4 processes according to the partition. The `LOCAL` vertices of each process strictly correspond to a previously defined subset of P_4, represented as plain vertices: each color is associated to a process. A set of dashed vertices is added, according to the edges of the initial graph, to preserve each vertex neighborhood. Such vertices can belong to any other subset of P_4, and so can be located on any other process, to form the set of `DISTANT` vertices U'_i. Finally, all edges between `LOCAL` vertices or between a `LOCAL` vertex and a `DISTANT` vertex are represented to constitute the local representation G_i of the distributed graph DG_N. Edges between two `DISTANT` vertices are not contained in G_i.

B. Properties

Let $G = (V, E)$ be a graph, P_N a partition of G and $DG_N = \bigcup_{i=1}^{N} G_i$ the associated distributed graph. The following properties ensure that the global structure of the graph G is completely preserved when it is transformed into the

`DistributedGraph` DG_N, whatever the partition P_N is. Properties 1 and 3 show that G can be built from DG_N, for any distribution of the graph. Moreover, property 2 shows that the direct neighborhood of any `LOCAL` node is preserved for any graph partition, what is a key feature of FPMAS.

Property 1. $\bigcup_{i=1}^{N} V_i = V$

Demonstration. *By definition, it is clear that* $\forall i \in [1, N], V_i = U_i \cup U_i' \subset V$ *so* $\bigcup_{i=1}^{N} V_i \subset V$. *Moreover, since* $\forall i \in [1, N], U_i \subset V_i$ *and* $P_N = \bigcup_{i=1}^{N} U_i = V$, $V \subset \bigcup_{i=1}^{N} V_i$ *so* $\bigcup_{i=1}^{N} V_i = V$. □

Property 2. $\forall v \in U_i, \exists w \in V, e = (\overline{v, w}) \in E \implies e \in E_i$

Demonstration. *By definition,* $w \in V$ *so* $\exists j \in [1, N], w \in U_j$. *If* $i = j$, $v, w \in U_i \subset V_i$ *so* $e \in E_i$. *If* $i \neq j$, *by construction* $w \in U_i' \subset V_i$ *(and* $v \in U_j' \subset V_j$*) so* $e \in E_i$ *(and* $e \in E_j$*).*[1] □

Property 3. $\bigcup_{i=1}^{N} E_i = E$

Demonstration. *By definition of* E_i, *it is clear that* $\bigcup_{i=1}^{N} E_i \subset E$. *Then, let* $e = (v, w) \in E$. $\exists i, j \in [1, N], v \in U_i, w \in U_j$, *so according to property 2,* $e \in E_i$ *and* $e \in E_j$ *so* $E \subset \bigcup_{i=1}^{N} E_i$ *and finally* $\bigcup_{i=1}^{N} E_i = E$.□

C. From DistributedGraph to MAS

In practice, we can consider each `Agent` as the data of a `Node` and the behavior of the `Agent` as a `Task` attached to this `Node`.

The previous properties allow the FPMAS framework to completely abstract the distribution of a MAS simulation, since:

1) The complete model is preserved, whatever the distribution is: according to properties 1 and 3, all agents and links are preserved for any graph partition.
2) According to property 2, the neighborhood of each agent is preserved.

Let's consider a *task* t bound to a `LOCAL` *node* n. The task t is assumed to be executed on the process which owns n, and can operate on any node linked to n. Since `LOCAL` and `DISTANT` nodes share the same software interface that provide read / write features, the behavior of each agent can be implemented independently of the current location of the neighbors, abstracting any distribution related issues. It is the responsability of the underlying implementation of the interface — that does not need to be known by the final user — to decide how operations must be handled depending on the `LOCAL` or `DISTANT` state of each neighbor.

In consequence, the `DistributedGraph` structure allows agents to interact with other agents of their neighborhood **independently of the graph distribution**: agents are allowed to interact *across processes*.

Notice that the concept of `DISTANT` nodes at the scale of each process can be considered as a generalization of the Repast HPC's OLZ or D-MASON's AOI.

[1] v is represented as a `DISTANT` vertex on process j, and w is represented as a `DISTANT` vertex on process k. In consequence, e is represented as a `DISTANT` edge on process j and k.

IV. FPMAS: A Distributed MAS Simulation Platform

FPMAS is an open source C++ implementation[2] of a Distributed MAS Simulation platform based on the `DistributedGraph` structure, to which synchronization and MAS simulation facilities have been added.

A. Presentation

The FPMAS platform must be thought as an execution layer on top of which well-known and existing simulation tools, such as GAMA [12] or SARL [13], might be interfaced and adapted to provide high-level modeling techniques to the user, while completely abstracting distribution related issues. The objective of the project is **not** to provide a new MAS modeling standard. The corresponding architecture is presented figure 2.

The *Low-Level Layer* contains distribution specific softwares and libraries in charge of the program distribution and other parallel features, such as MPI or OpenMP. FPMAS then abstracts those features thanks to a synchronized and dynamic distributed graph structure, on which existing *MAS Modeling Toolkits* can interface to distribute agent models. The final user can implement agent-based models using simplified tools, without struggling with distribution issues, still allowing him to run his model on HPC resources.

The development of the platform is based on the design of interfaces (*dependency injection*) and the usage of *unit tests* and *mocks* (*test-driven development*) allowing software components design with a strict separation of concerns [21]. In consequence, each component can be easily re-implemented and switched without any alteration of the global system. Moreover, in some cases, multiple implementations of the same interface can be provided, or even user defined, to easily adapt and optimize a component for a given project, without

[2] https://github.com/FPMAS

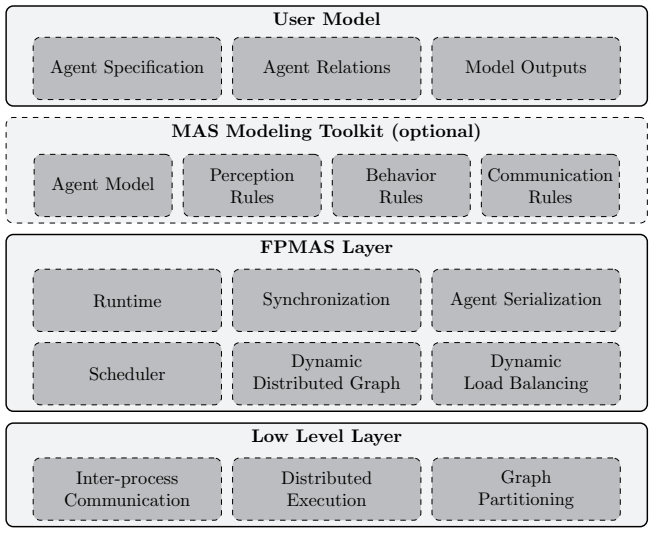


Fig. 2: Global FPMAS software stack

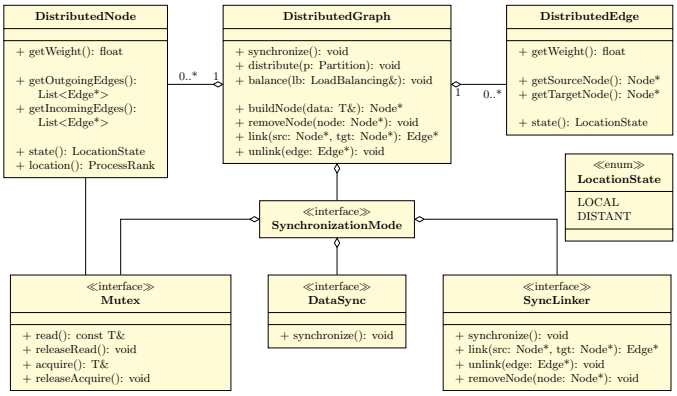


Fig. 3: FPMAS Graph Synchronization class diagram

altering the whole system. This might be very effective in HPC, to target and optimize components where performance bottlenecks are identified. Such a design is also very useful to reach a decent level of abstraction for the final user: since generic interfaces do not depend on their implementation, each feature provided by the API is straightforward and concise. The `LoadBalancing` interface is notably completely independent from graph distribution or synchronization features, what allows to easily interface with any existing load-balancing algorithms. FPMAS currently provides a `LoadBalancing` implementation based on Zoltan.

The core classes responsible for the FPMAS graph synchronization are represented on figure 3 and explained in the next sections.

B. Graph Synchronization

The synchronization policy is a key point in a distributed simulation since it may have a great impact on performance and simulation results. Moreover different simulations might have different synchronization needs, depending on their concurrency properties. For this reason, FPMAS introduces *graph synchronization* techniques to support two types of operations on the `DistributedGraph`:

- Data operations: allow to perform concurrent read / write requests on nodes' data.
- Dynamic graph management operations: allow to build and remove nodes and edges *at runtime* from any process.

Synchronization policies are enforced using the `SynchronizationMode` interface, compound of three generic software components:

- `Mutex`: are used to manage access to each node's data.
- `DataSync`: synchronizes data operations.
- `SyncLinker`: synchronizes dynamic graph management operations.

As detailed in the next sections, the specification of those components is voluntarily as much generic as possible to support a wide range of synchronization modes, through different implementations of each component. This was notably required to support at least two synchronization modes:

- `GhostMode`: Corresponds to the RepastHPC OLZ synchronization, without writing between processes.
- `HardSyncMode`: An FPMAS specific synchronization mode, that **allows read / write operations** between processes.

C. Mutex

A `Mutex` instance is associated to each `Node`, `LOCAL` and `DISTANT`, and is used to access node's data.

`Mutex` methods are available to the user as a generic interface: abstract `read()` and `acquire()` operations can be performed, independently of the current `SynchronizationMode`. The actual behavior is determined once a `SynchronizationMode` implementation is *injected*, i.e. provided by the user as a `DistributedGraph` parameter.

No modification can be performed on *read* data. However, it is possible to call **any method** on *acquired* data, including ones modifying the data. In any case, each operation must be released.

Notice that how each operation is handled depends on the implemented `SynchronizationMode`. In consequence, *read* and *acquired* data are **not** guaranteed to be up to date. Write operations on `DISTANT` nodes and concurrency management are not required at the generic `Mutex` component level. Some implementation might forbid `DISTANT` write operations for performance purpose for example.

D. DataSync and SyncLinker

The abstract `DataSync` and `SyncLinker` `synchronize()` methods are used by the `DistributedGraph` synchronization and distribution methods to implement the simulation steps.

Each `SyncLinker` operation can apply to `LOCAL` **and** `DISTANT` nodes, and must be committed at the next `synchronize()` call, whatever the implemented `SynchronizationMode` is. Those methods are not supposed to be called directly by the user, but are implicitly used by the corresponding high-level `DistributedGraph` methods.

E. Operations scope

Considering a *task* t bound to node n, t can **only** apply the `Mutex` and `SyncLinker` operations to nodes located in the neighborhood of n, including n. Considering the theoretical properties of Section III, this ensures that the execution of any task t is independent from the graph distribution, i.e. from the location of nodes in the neighborhood of n.

F. Synchronization Modes

FPMAS currently supports two synchronization modes: `GhostMode` and `HardSyncMode`. More synchronization modes might be defined in the future for performance purpose, to enable multi-threading or to support model specific requirements for example.

In any case, the final user is not required to deal with `SynchronizationMode` features and implementation: only generic `read()` or `acquire()` operations are performed. The `SynchronizationMode` is just a meta-parameter

specified at the model scale. This allows for example to produce robust and reliable benchmarks, since the synchronization techniques and so the global execution logic can be changed **without modifying the model implementation**. However, it might be interesting for the user to know the limitations and basic behaviors of each `SynchronizationMode` to understand impacts on results and performances.

1) Ghost Mode: The `GhostMode` synchronization mode directly corresponds to the synchronization performed by Repast HPC:

- `DISTANT` nodes' data returned by `read()` and `acquire()` methods are only updated at each `DataSync::synchronize()` call, which means that their state is that of the previous step (ghost).
- Modifications performed on *acquired* data of `DISTANT` nodes are **not** committed to the distant processes.
- `link()`, `unlink()` and `removeNode()` operations are committed at each `SyncLinker::synchronize()` call.

2) Hard Sync Mode: The `HardSyncMode` synchronization is a major improvement in the field of distributed MAS simulation, since it enables *read* and *acquire* operations across processes, with a true concurrency management. Each time `read()` and `acquire()` calls are performed on a `DISTANT` node, up to date data is fetched from the origin process and locks / shared locks are applied according to the *first readers-writers problem* [22]. Moreover, each method blocks until data is available, even if data is `LOCAL`: a process might wait for other processes to release its own data before it can access it itself. `link()`, `unlink()` and `removeNode()` operations are also committed *on the fly*.

`DataSync` and `SyncLinker synchronize()` methods implement a `TerminationAlgorithm` [23] to wait until all requests are handled.

V. Experimentations

The following experimentations demonstrate the usage of the FPMAS library to implement a simple graph based MAS model even if some features are still required to easily handle spatial MAS. The example has voluntarily been designed to require write operations across processes to run properly, what is allowed by FPMAS but would be impossible with other classical platforms. A brief study about performances is also introduced.

Experimentations was performed on the computing facilities of the *Mesocentre de calcul de Franche-Comte*[3], and represent more than 9000 hours of cumulated CPU time. It is interesting to note that no deadlock, memory leaks or other issues were observed during all the simulation time, for the two synchronization modes. Such a robustness is clearly due to the strict unit testing policy under which the platform is developed.

The complete implementation of the example is available on GitHub[4].

[3]http://meso.univ-fcomte.fr/

[4]https://github.com/FPMAS/fpmas-sir

A. SIR model

Examples presented in this section are based on the "metapop" model introduced in [24]. The model represents the evolution of a virus in a set of cities, based on the SIR model.

A given number of `City` agents are initiated as nodes of the graph. Cities are implicitly and uniformly distributed in a 2D space. Each city is then connected to its k **nearest neighbors** to build a *clustered* graph, so that the number of outgoing edges of each city follows a Poisson distribution of parameter $\lambda = K$, parameter of the model.

The *population* of each city is represented by three values:

- S (susceptible): number of people that might be infected by the virus.
- I (infected): number of people currently infected, who can infect susceptible individuals.
- R (removed): number of people removed from the infected people (by recovery or by death). Removed people cannot be infected twice.

The total population of the city is hence $N = S + I + R$.

Each city uniformly sends a proportion $g = 0.12$ of its population to its neighbors at each time step.

A `Disease` agent is connected to each `City`. Its purpose is to update the city population according to the following equations:

$$\begin{cases} \frac{dS}{dt} = -\frac{\beta IS}{N} \\ \frac{dI}{dt} = \frac{\beta IS}{N} - \alpha I \\ \frac{dR}{dt} = \alpha I \end{cases}$$

α and β parameters are associated to each `Disease` instance. More particularly, α represents the recovery rate and β characterizes the virus transmission rate among the population. In our experimentation, we use the constants $\alpha = 0.2$ and $\beta = 0.5$.

At each time step, each `Disease` agent *writes* population updates to its connected `City`, solving the previous differential equation system using the *Range-Kutta 4* method.

A time step of the model is described as follow:

1) `City` agents execution (performs population migrations)
2) Global graph synchronization (ensures all migrations have been performed)
3) `Disease` agents execution (updates population of each city)
4) Global graph synchronization (ensures all populations are updated)

B. GhostMode vs. HardSyncMode

An important aspect of the model is that each type of agent is performing *write* operations on its neighbors. More particularly, cities are concurrently writing to their neighbors: while a city is migrating its own population to others, others are migrating population to it, at each time step.

The definition of the model implies that the global population is constant. But depending on the synchronization mode, *write* operations performed on cities contained in `DISTANT`

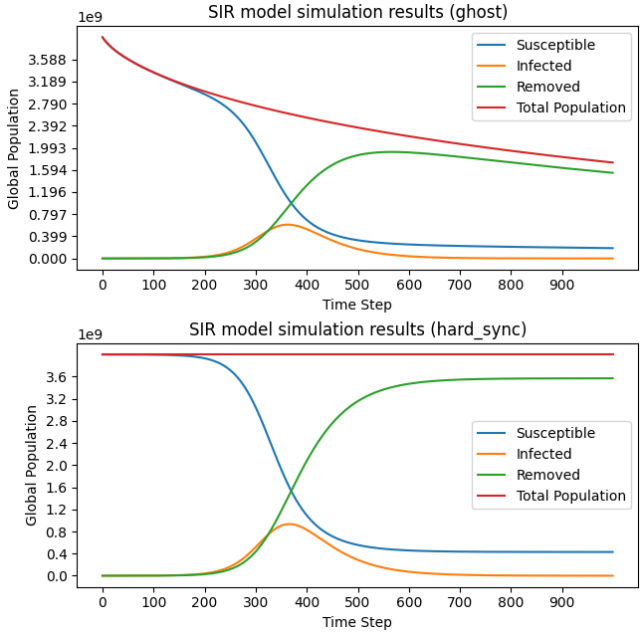


Fig. 4: Results obtained by the simulation of the SIR model with 100,000 cities and K=12, on 64 processes

nodes might not be reported to the origin process, what would cause a diminution of the global population. Notice that this behavior corresponds to a default implementation performed with Repast HPC, and also to the FPMAS `GhostMode`. As a reminder, it is possible to switch the FPMAS synchronization mode from `GhostMode` to `HardSyncMode` using a single meta-parameter, without altering the implementation of agents. The differences of results, when the model is run with 100,000 cities and a relatively high connectivity degree $K = 12$ on 64 processes are presented on figure 4.

It is clear that results obtained in `GhostMode` are significantly altered. Indeed, each city has a high probability to be linked to at least one `DISTANT` city, what causes population "leaks" from this city at each time step.

On the contrary, the `HardSyncMode` produces results **as robust as the sequential execution**, compared to simulations executed on one process, and to the well known SIR model result curves. Each of the 100,000 cities are concurrently writing to 12 other cities in average, potentially on `DISTANT` nodes, executed on 64 parallel processes, but every single write operation is performed, without any race condition or deadlock issues. Moreover, no distribution related problems was considered from the user point of view: the only requirement is to `acquire()` the city on which data is written.

C. Performances

Here we present some brief performance results, as a proof of concept. Execution times for different values of K are presented on figure 5. Values presented are average execution times of 5 executions: observed variations between executions are too small to be represented as box plots.

The test model presents some great speed ups from 1 to 16 processes, but does not improve from 16 to 64 processes, due to the computation / communication ratio. For 10,000 cities, what is relatively small, the communication cost is such that execution time raises for more than 16 processes in `GhostMode`. Indeed, since the graph is *clustered*, the automatic load-balancing algorithm efficiently distribute the model over available processes, but increasing the number of processes inevitably raises the time consuming communication volumes.

Experiments led with more constrained unclustered graph shapes even show that the distribution of the model might produce larger global execution times than sequential executions, due to high communication volumes between processes. Some profiling analysis might help to identify the real communication costs.

It is also interesting to note that even if the `GhostMode` is more efficient than `HardSyncMode` for small number of processes, the curves tend to converge for large number of processes, what might imply that allowing concurrent write requests across processes does not necessarily produce a significant loss of performances.

In any case, those preliminary results clearly show the need for a more complete and significant FPMAS performance benchmark to compare synchronization modes on several models according to different agent dependencies levels.

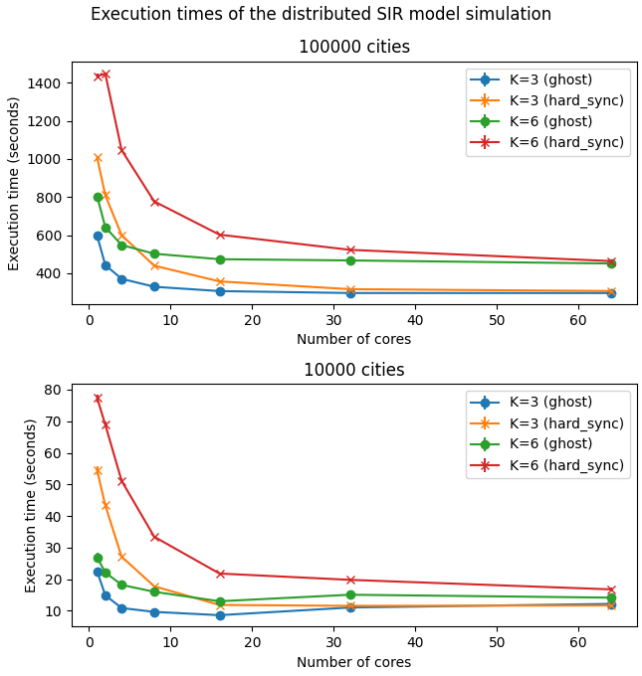


Fig. 5: Execution times for the SIR model, using a *clustered* graph

VI. Conclusion

In this work, we introduce the concept of a `DistributedGraph` structure and demonstrate some useful properties

that allow the efficient and implicit distribution of a graph over a set of processes, without losing any information about the original graph. Such a distribution allows tasks bound to nodes to perform read and write operations on data of neighbor nodes, as well as dynamic graph management operations such as link, unlink, node creation and removal, independently of the underlying distribution of the graph. This allows to completely abstract the distribution process to the final user, and to interface with any automatic graph load-balancing algorithm. Some generic *Synchronization modes* have been introduced. The `HardSyncMode` notably allows concurrent write requests across processes, what is a feature provided by none of the surveyed distributed MAS platforms. Experimentations based on a SIR model show the importance of such distant write operations on the model results. Performance analysis was also led, and suggest further studies to analyze distribution limits on some kind of MAS simulations as well as on the influence of synchronization modes. Future works also includes a better support for a generic spatial MAS, to reach a new level of abstraction of the underlying graph and to allow an easier interfacing with existing MAS simulation platforms.

ACKNOWLEDGMENT

This work was partly supported by the French Research Agency under the EIPHI Graduate School (contract "ANR-17-EURE-0002") and the Franche-Comté region under the project SYMPAMA (contract CRBFC:2019-Y-09120).

REFERENCES

[1] P. Seppecher, "Flexibility of wages and macroeconomic instability in an agent-based computational model with endogenous money," *Macroeconomic Dynamics*, vol. 16, pp. 284–297, Sept. 2012. Publisher: Cambridge University Press.

[2] E. Blanchart, N. Marilleau, J.-L. Chotte, A. Drogoul, E. Perrier, and C. Cambier, "SWORM: an agent-based model to simulate the effect of earthworms on soil structure," *European Journal of Soil Science*, vol. 60, no. 1, pp. 13–21, 2009.

[3] A. Banos, N. Corson, B. Gaudou, V. Laperrière, and S. R. Coyrehourcq, "Coupling Micro and Macro Dynamics Models on Networks: Application to Disease Spread," in *Multi-Agent Based Simulation XVI* (B. Gaudou and J. S. Sichman, eds.), Lecture Notes in Computer Science, (Cham), pp. 19–33, Springer International Publishing, 2016.

[4] N. Wagner and V. Agrawal, "An agent-based simulation system for concert venue crowd evacuation modeling in the presence of a fire disaster," *Expert Systems with Applications*, vol. 41, pp. 2807–2815, May 2014.

[5] M. Pipattanasomporn, H. Feroze, and S. Rahman, "Multi-agent systems in a distributed smart grid: Design and implementation," in *2009 IEEE/PES Power Systems Conference and Exposition*, pp. 1–8, Mar. 2009.

[6] N. Collier and M. North, "Parallel agent-based simulation with Repast for High Performance Computing:," *SIMULATION*, Nov. 2012.

[7] M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, and C. Greenough, "FLAME: simulating large populations of agents on parallel hardware architectures," in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '10, (Toronto, Canada), pp. 1633–1636, International Foundation for Autonomous Agents and Multiagent Systems, May 2010.

[8] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo, "Bringing together efficiency and effectiveness in distributed simulations: The experience with D-Mason," *SIMULATION*, vol. 89, pp. 1236–1253, Oct. 2013. Publisher: SAGE Publications Ltd STM.

[9] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, "Complex adaptive systems modeling with Repast Simphony," *Complex Adaptive Systems Modeling*, vol. 1, p. 3, Mar. 2013.

[10] U. Wilensky, "Netlogo," 1999.

[11] F. Michel, "The IRM4S model: the influence/reaction principle for multiagent based simulation," in *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, (Honolulu, Hawaii), pp. 1–3, Association for Computing Machinery, May 2007.

[12] P. Taillandier, B. Gaudou, A. Grignard, Q.-N. Huynh, N. Marilleau, P. Caillou, D. Philippon, and A. Drogoul, "Building, composing and experimenting complex spatial models with the GAMA platform," *GeoInformatica*, vol. 23, pp. 299–322, Apr. 2019.

[13] S. Rodriguez, N. Gaud, and S. Galland, "SARL: A General-Purpose Agent-Oriented Programming Language," in *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, vol. 3, pp. 103–110, Aug. 2014.

[14] A. Rousset, B. Herrmann, C. Lang, and L. Philippe, "A survey on parallel and distributed multi-agent systems for high performance computing simulations," *Computer Science Review*, vol. 22, pp. 27–46, Nov. 2016.

[15] N. Collier, J. Ozik, and C. M. Macal, "Large-Scale Agent-Based Modeling with Repast HPC: A Case Study in Parallelizing an Agent-Based Model," in *Euro-Par 2015: Parallel Processing Workshops* (S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander, eds.), Lecture Notes in Computer Science, (Cham), pp. 454–465, Springer International Publishing, 2015.

[16] P. Richmond and M. K. Chimeh, "FLAME GPU: Complex System Simulation Framework," in *2017 International Conference on High Performance Computing Simulation (HPCS)*, pp. 11–17, July 2017.

[17] A. Rousset, B. Herrmann, C. Lang, L. Philippe, and H. Bride, "Using Nested Graphs to Distribute Parallel and Distributed Multi-agent Systems," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 710–717, Feb. 2016. ISSN: 2377-5750.

[18] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent Advances in Graph Partitioning," in *Algorithm Engineering: Selected Results and Surveys* (L. Kliemann and P. Sanders, eds.), Lecture Notes in Computer Science, pp. 117–158, Cham: Springer International Publishing, 2016.

[19] 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 *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–11, Mar. 2007. ISSN: 1530-2075.

[20] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, vol. 26, pp. 1519–1534, Nov. 2000.

[21] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, (Vancouver, BC, CANADA), pp. 236–246, Association for Computing Machinery, Oct. 2004.

[22] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"," *Communications of the ACM*, vol. 14, pp. 667–668, Oct. 1971.

[23] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren, "Derivation of a termination detection algorithm for distributed computations," in *Control Flow and Data Flow: Concepts of Distributed Programming* (M. Broy, ed.), Springer Study Edition, (Berlin, Heidelberg), pp. 507–512, Springer, 1986.

[24] A. Banos, N. Corson, B. Gaudou, V. Laperrière, and S. R. Coyrehourcq, "The Importance of Being Hybrid for Spatial Epidemic Models:A Multi-Scale Approach," *Systems*, vol. 3, pp. 309–329, Dec. 2015. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute.