CoeGSS
Centre of excellence

D3.8 – Report of framework for prototyping of parallel Agent Based Modelling Systems

Grant Agreement	676547
Project Acronym	CoeGSS
Project Title	Centre of Excellence for Global Systems Science
Topic	EINFRA-5-2015
Project website	http://www.coegss-project.eu
Start Date of project	October 1, 2015
Duration	36 months
Deliverable due date	30.09.2018
Actual date of submission	30.09.2018
Dissemination level	Public
Nature	Report
Version	2.0 (after internal review)
Work Package	3
Lead beneficiary	GCF
Responsible scientist/administrator	Andreas Geiges
Contributor(s)	Sarah Wolf, Gesine Steudle, Steffen Fürst,
Internal reviewers	Sergiy Gogolenko Daniela Paolotti
Keywords	ABM-modeling, parallel, framework,
Total number of pages:	22

The CoeGSS (“Centre of Excellence for Global Systems Science”) project is funded by the European Union. For more information on the project please see the website http:// http://coegss-project.eu/

Version History

	Name	**Partner**	**Date**
From	Geiges, Andreas	GCF	21.9.018
First Version	for internal review		Sept. 21, 2018
Second Version	for submission		Sept. 26, 2018
Reviewed by	Sergiy Gogolenko Daniela Paolotti	HLRS ISI	Sept. 26, 2018 Sept., 25, 2018
Approved by	Coordinator	UP	Sept. 28, 2018

Abstract

This deliverable presents ABM4py, an open source framework for prototyping agent-based models (ABM) with explicit support for spatial domains, (social) network structures and parallel execution. ABM cannot fit in a certain class of computer models, since the concept of interacting agents can in principle be used to model many (social or natural) systems. In recent years, ABM gained importance in the social sciences and, in particular, in Global Systems Science, where agent-based models are applied to explore potential evolutions in view of global challenges and systemic risks. In the very heterogeneous field of social sciences, the flexible structure of ABM is of great interest and many ABM frameworks and models have been developed. Current ABM often are limited by computing resources, and thus there is a new community that aims at using high-performance computing (HPC) in social sciences in the future. However, HPC evolved from numerical modeling of natural systems, solving differential equations using numeric approximations and traditionally relies on exploiting these specific model structures. Since models in social sciences, and especially agent-based modeling is lacking a common structure, is very hard if not impossible to design general-purpose high-performance solutions. Thus, for mature models, a specific HPC implementation of the model is usually required at a certain state of development. However, multi-purpose prototyping ABM frameworks that allow for rudimentary parallel execution are often missing among the various ABM frameworks. Repast-HPC is one of the few candidates for parallel ABM frameworks. However, parallel execution, facilitated to a non-(HPC)-expert, would allow social scientists to create sophisticated ABM models. Thus, the purpose of this open-source framework is to bridge the gap between prototyping and HPC in the development of ABM for social sciences. ABM4py is implemented in python and therefore allows using libraries from a broad python ecosystem, covering the areas of machine-learning, geo-information systems, data sciences and more. ABM4py uses a graph-based approach and hides the parallelization for non-experts, such that models of several millions of interacting agents can be executed on up to several hundred processes.

Table of Contents

1 Introduction

This document introduces ABM4py, an open source framework for prototyping agent-based models (ABM) with explicit support for spatial domains, (social) network structures and parallel execution. The framework is available under the GNU Lesser General Public License V3 and available at: https://github.com/CoeGSS-Project/abm4py.

1.1 The purpose of combining ABM and HPC in python

When thinking about high performance computing languages, python usually does not come to mind. This section will emphasize why a python-based framework nevertheless has its place in the development of models in social sciences, and especially when bridging the ABM and HPC communities together. Murphy (2014) states that a "simulation created in Repast HPC will never compete with a special-purpose application. Once a simulation's dynamics are established, a pure MPI version will be possible and will run faster (probably much faster). However, the MPI version will almost certainly lack the flexibility of Repast HPC [North2013]; it will be an Agent-Based Model, but it will not help with Agent-Based Modeling." This statement implies two things. (1) Since each model determines the structure, prototyping a new ABM seldomly can rely on existing structures and efficient (tailored) HPC-implementation. (2) The possible later implementations of a parallel HPC code do not provide general building block in general. This is mainly the case due to the flexibility of ABM, their lack of common structure and wide application range. Thus, overall prototyping will always be an essential first stage in the development of large-scale parallel ABM for social sciences.

At the same time, the landscape of social modeling differs quite a lot from the conventional approaches to modeling in earth science. Complex social models cover a wide range of fields including geo-information science, (big-)data science, machine learning, statistics and social behavioral models and possibly connect to existing earth science models. The modular and open-source focus of the python language provides a modular system of libraries that cover almost any purpose in the above mentioned fields and thus enables faster implementation of prototypes.

Thus, the purpose of this ABM framework is to provide a library or toolkit for prototyping ABM that bridges the gaps between the flexibility of ABM, the open-source toolbox of the python ecosystem and the resources of HPC. Modelers in social sciences, most of whom are not experts in parallel programming, should nevertheless be able to implement parallel prototypes of large-scale ABM for the purpose of testing and refining them in an iterative process. As mentioned, special purpose implementations will always be more efficient than implementations based on frameworks – ABM4py shall allow modelers to arrive at the point where they can decide that the prototype ABM "works" sufficiently well to be translated into a special purpose implementation as a next step.

1.2 Approach

ABM4py is not a traditional framework in a sense that the model code is not executed inside the framework. As mentioned above, it is rather a toolbox, providing pre-defined classes, functions and a dynamic and efficient data structures. All classes serve as templates to derive more complex classes, which extend the base functionalities. In this way, ABM4py provides basic functionalities, but does not restrict the flexibility of the ABM approach.

We identified a directed *multi-graph* as the most fitting data structure for agent-based modeling. In this graph, agents (micro-level entities) are represented in the graph as nodes, and relations between agents as links between agents. To model different types of agents and relations, the graph consists of different node and link types, which differ in their attributes and methods. Each agent is only allowed to access and alter attributes of its own node and of outgoing edges, which we define as the agent's scope. Detailed information about the graph implementation is given in section 3.4.

The `World` class is the overarching structure of the framework and provides the ABM environment with the functions necessary to set up the different components of the simulation, manages registers of agents, grants global access to all agents, edges and their attributes and schedules the simulation. In case of parallel execution "all" refers only to the agents existing on the process (living agents) and copies of agents from other processes (ghost agents). Multiple subclasses structure the functionalities of the environment like parallel communication, data storage (`Graph`), spatial grid, input / output (`IO`), random module and a distributed statistics module. Section 3.2 provides more details of the world as an environment and control structure.

The defining feature of ABM is that the overall (or macro-level) behavior (or dynamics) of a system under study is observed as it arises from the interplay of many events at the micro-level. There are wider and more restricted uses of the term "agent" in this setting. We loosely follow the description that accompanies Overview, Design concepts and Details (ODD) that classifies "types of entities", but refers to agents, since speaking of entity-based models would take it too far and thus, distinguish between agents/individuals, spatial units, environment, and collectives. To avoid deep definition issues we interpret agents to be the "players" of interest, as they might be called in describing the real-world global challenge under study or in game theory and which populate the model in high numbers.

The `Agent` class is a template class for more complex agent types. It provides the basic methods to implement micro-dynamics or transition functions and state variables. The available methods of the agent are in line with scope of the agents (see Section 3.1). The basic agent is described in detail in Section 3.3.

Additional traits for adding additional functionalities can be used when deriving new agent classes. Examples for such higher-class agents is a Traveler that is an agent with the additional trait `Mobile` or a `Location`, which is an agent with the trait `Gridnode` that provides it with spatial relationship functions to its grid neighbors. The currently implemented traits are introduced in section 3.5.

Currently parallel distribution of the simulation is implemented by partitioning the spatial dimension of the graph only. This is the case because local conditions or spatial proximity) play an important role in the type of models for which ABM4py was developed. Extensions to general graph partitioning are possible and we invite users to extend this open source tool with further more general partitioning as needed to fit their models. A theoretical concept for a more general approach is presented in the GCF Working Paper “Distributed agent graph”. [Fuerst 2018 (Working paper)] The general concept of the parallel communication will be explained in detail in section 3.7, which relies on the limited scope of individual agents and ghost copies of agents.

2 Features of Global Systems

“The vision of Global Systems Science (GSS) is to provide scientific evidence and means to engage into a reflective dialogue to support policy-making and public action and to enable civil society to collectively engage in societal action in response to global challenges like climate change, urbanisation, or social inclusion. GSS has four elements: policy and its implementation, the science of complex systems, policy informatics, and citizen engagement” (Dum+ Johnson2017). To support policy making in this field and other related fields, the relevant system parts of interest need to be described in a model.

The systems to be represented by ABM in this context can be described in terms of an individual, a social and two environmental layers (see also ref D4.4). Agents decide/act based on

- individual situation and individual appraisal of options, e.g., expected outcomes of alternative actions
- social interaction, and information gathered via different networks (contact network with face-to-face interactions, social networks, kinship relations etc.)
- their environment. This contains “global” elements such as the media, advertisement, the law, regulations or policies, as well as local conditions in a geographical dimension.

Just from the agents’ perspective, both social networks and global influence factors, as well as local environments may dynamically evolve over time, and there is usually a mutual influence between agents’ actions and the evolution of networks and environments. Network structures can be complex, involving hubs, clusters, hierarchical and community structures.

Agents and spatial layers are supposed to reflect the real world more closely than it is the case for ABM created to study mechanisms of social interaction in more abstract or theoretical settings. Therefore, synthetic populations and maps, that is, geographically heterogeneous information, are among the inputs to ABM for GSS.

3 Framework structure

3.1 Basic concepts

In ABM4py, agents are classified by different types, which are normally implemented as a new python class, derived from a base agent class. The class defines (a) the agents state variables including dynamic and static variables and (b) the actions of agents of this type, which are implemented as class methods. The complete definition of agents and the organization of different types of agents is described in section 3.3. Agents and their interactions are modeled as directed graph, in which agents correspond to nodes and interactions correspond to edges. Both, nodes and edges may contain state variables. Each agent has a limited scope of interaction and information, which is defined by its outgoing edges (see Figure 1. In the basic case, an agent can access global variables, outgoing edge state variables and read state variables of agents connected by outgoing edges. The scope should not be mixed up with the interaction radius, which defines the maximum distance to introduce interactions. Thus, by adding or removing connections, the scope can change, but only within the interaction radius. More details about the graph and its different purposes can be found in section 3.4.

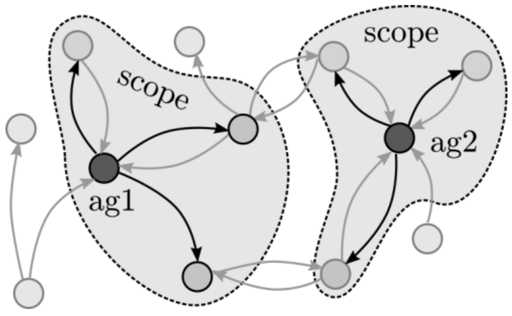

Figure 1: Visualization of the agents' scope modeled as a directed graph

Connection between agents (or graph edges) are of different types as well. Each edge type connects one particular agent type to another, thus you can infer the source and target agent type using the edge type. This has many implications for modeling and performance, for example the group of agents connected to an agent by a certain type is of a single type and, thus all have the same state variables. This allows to perform vectorized access for many agents within the agents scope.

ABM4py provides two basic template agents (see also section 3.3), `Agent` and `Location`. `Location` serves as an example of an agent type that is related to the spatial dimension. Furthermore, it provides different traits that can be used to amend the base agent classes. A basic example is the trait `Mobile`, which allows an agent to change its spatial position (see examples folder) .

The next section introduces the class `World` which is often called the environment in agent-based modeling. In ABM4py, it also functions as the main interface for the modeler to control and access all components of the model.

3.2 World class (Environment)

The world class is the overarching class that connects all components of ABM4py and has three main functions. Firstly, the world instance is the register and scheduler for agents and contains global variables, parameters and other control elements. Secondly, it provides the main functional user interface for the modeler, which is used for accessing and controlling all components of the simulation. Thirdly, it is the global environment in the sense of "forces that drive the behavior and dynamics of all agents or grid cells". For comparison, Pandora has a "world" which "manages the different layers of continuous information that define the environment of the simulation" . Thus, global environments (there may be more than one) representing a world instance may also have a mutable state, and can collect or broadcast information from or to all agents in the graph, without explicit connections to and from all agents.

In a complex model, the world class of ABM4py functions as a base class from which more sophisticated environments can be derived. After initialization, the world provides the basic user interface for

- registering new agent types and creation of agent instances
- registering new connection types used to connect agents
- registering new global variables
- initialization of utility sub-classes that simplify different workflows
- global access of register agents, their connection or their attributes
- vectorized access and alteration of agent attributes

Several utility sub-classes are available to facilitate different common workflows in ABM. The `IO` sub-class provides an easy interface to store the current state of the agents and edges of particular types. Currently, hdf5 files format is used for the output of agent and connection attributes, since it also supports parallel IO. Furthermore, the interaction graph can be stored as an adjacency file, which can be used as an interface to, for example, graph partitioning software (e.g. metis).

The `Grid` class facilitates the generation of regular grids, based on array-like input files which provide the geometry. It is coupled with the use of the agent class `GridNode` or classes derived from it. For example, an automatic function can be used to connect grid nodes in a defined interaction radius (see example below). The generated gridnodes are agent instances themselves and can be used in the further simulation.

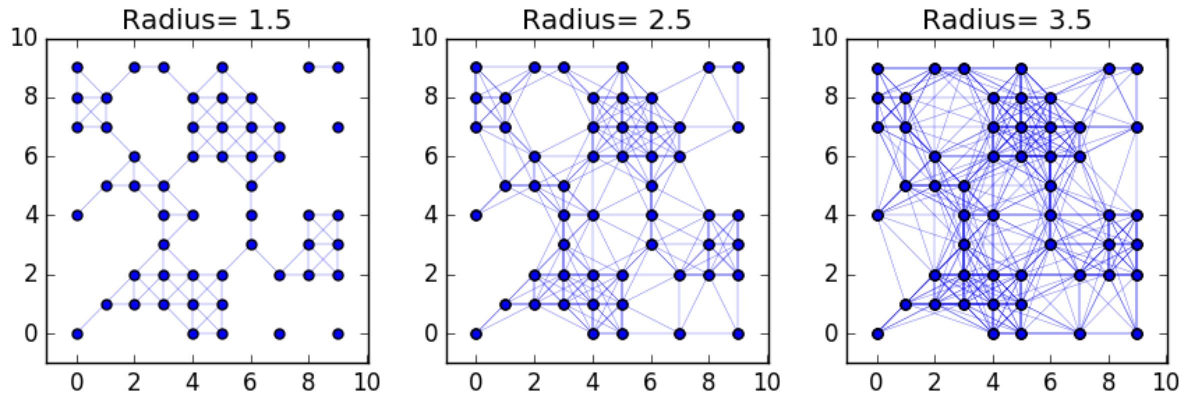

Figure 2 :Example for the generation of a spatial layer. The nodes represent grid nodes and the lines the connections.

In the current version, the grid is also strongly related to the parallelization of the simulation, since the partitioning currently requires a spatial domain. More details on the parallelism is given in section 3.7.

The class `Random` helps the modeler to implement the important randomization into the model. At the current state, it allows to access agents of a specified type in a random order or a randomly selected subset of specified size. Other functionalities will be added in the future.

The next two subclasses are related to the parallelism of ABM4py and help the user to organize the execution of the simulation on multiple processes. More details about parallel execution can be found in section 3.7.

The `PAPI` (parallel agent passing interface) class organizes the exchange and communication between processes in parallel execution. This includes the initial communication based on a given spatial grid and its partitioning, the transfer of ghost agents and the update of ghost agent attributes. Ghost agents are copies of agents on distributed processes and will be explained in section 3.7.

The `globals` class helps the user to manage global variables and statistics. Thereby, global variables are set or computed by the modeler and then aggregated and shared between all processes, whereas global statistics are related to attributes of agents. This allows to compute statistics of different attributes over multiple parallel processes, for example the average age of all agents or the standard deviation of the income of all agents.

3.3 Agents

The agent itself is the core unit and purpose of existence of ABM and provides a large flexibility to the modeler as a concept. Thus, an agent can be almost everything including atoms, persons, animals, companies, or economies, and it can be defined by a set of state variables and transition functions, i.e. functions to change their states based on the current and/or previous states. In addition, agents might have a limited scope of access to the state variables of other agents and connections between them. This is one of the major foundation of ABM according to Ebstein 1999. ABM4py implements this interactions as outgoing connections to agents, which then become “peers”. After establishing a connection, the state of the outgoing connection can be seen as an extended agent state and the state of the peer becomes readable.

The class `Agent` provides the functions for accessing this scope and can be used to derive new agent abilities. The core abilities of the base agent are:

- adding/removing/changing links
- accessing attributes of links and peers
- accessing a list of peer IDs or link IDs
- accessing the peers itself
- counting peers

To extend this set of abilities, new classes are supposed to be derived from this class which define an extended state space and new methods. An example for this creation of new agent classes can be found in *examples/03_tutorial_wolf_sheep_grass.py* , where an agent type `Wolf` is derived from the base class `Agent` and the additional trait `Mobile` which allows the “wolves” to move in the spatial domain. The “wolves’’ state variables are extended by a “weight”, which is done in the `__descriptor__` function, which optionally also defines the type and length of the state variables. Thus, state variables can also be vectors, like a coordinate, consisting of x and y. New abilities are defined as methods like “hunt” and “move” are defined, where as move is a re-definition of function provided by the trait “mobile”. In the example as step function is used to schedule all the action of the class wolf.

Currently, not all traits can be combined (for example mobile and parallel), due to current limitations of the parallel implementation.

After a new class is defined, actual agents (i.e. instances) can be created with different states, which then establish a population of heterogeneous agents (another foundation of ABMs (Ebstein 1999). Each agent instance additionally needs to be registered to the world in order to become a part of the simulation.

The agent type `Location` extends the base agent by the trait grid-node and therefore represents the spatial extent. They can be organized in an arbitrary grid, the use of a regular grid is supported by automated generation functionalities (see section 3.2). However, any other organization structure can be implemented by the user. Connections between grid nodes are weighted with weights that are inverse proportional to the distance measure.

The assignment to a location of other entities is represented via an edge of a specific type. (Grimm 2010) point to a potential overlap of roles of spatial units, as entities with their own variables or as location attributes of other agents. Locations are an example for multiple possible roles within a simulation, they can function as spatial representatives, aggregators or collectives. Thus, locations can also be used to iterate over or aggregate information from all other entities connected with them. In the future, aggregators and collectives might be implemented as an additional trait to facilitate and speed up aggregation.

3.4 Graph (Data structure)

This section describes the graph as the underlying data storage structure. In general, ABM4py aims at hiding the graph completely from the modeler, so that the graph is only accessed by the given user interface. However, the information about the underlying structure is very handy for writing fast code and finding bottlenecks. Furthermore, some more sophisticated implementations might require a more direct access to the state space as given by the ABM4py-UI.

As mentioned before, agents in ABM4py correspond to nodes in the underlying directed graph and different agent types translate to different node types. The graph is the main storage structure of all state variables, except for the global environment.

The states are saved as an array with named columns for ach attribute (state variable) and one row per each agent. Thus, different agent types are stored in different arrays and for allowing for different sets of state variables.

Several identifiers (IDs) are used to identify agents and links. Each agent type is enumerated as a type ID (*agTypeID*), which is assigned when a new agent type is registered. When generating the individual agents, each agent receives a data ID (*dataID*), which is the row index in the attribute array and a local ID (ID), which encode both, *agTypeID* and dataID to a unique number. In case of parallel execution of the simulation, additionally a global ID (*glID*) is generated, which is unique over all processes.

Link IDs behave similarly, however a global ID is not provided as links are not shared over multiple processes (see section 3.7). In addition to agents, the state space of links is extended by a source ID and a target ID.

For many models, the class of a hierarchical graph often applies, where fewer spatial locations organize different kinds of agents on top. In this view, it might be helpful to picture links between agents of the same type as horizontally aligned and links between different types as vertical links. Figure 3 illustrates an example of a hierarchical graph, consisting of three agent types. The grid nodes organize the spatial structure, households are connected to the grid nodes which again are the connecting units for several persons.

Saving the states of all agents in a central storage has mainly performance reasoning connected to faster access and alteration of values. Therefore, the world as the environment with global access, IO, and aggregations can be speeded up significantly. The access from the view point of individual agents is mapped with references to the respective part of the state attribute array (see section 4.2). Overall, the graph allows for a dynamic and flexible data structure from the perspective of the individual agent, while the underlying array allows for faster vectorized access from a global view.

Finally, the graph also stores the individual instances of the agents so that they can be accessed quickly by the UI methods and iterators.

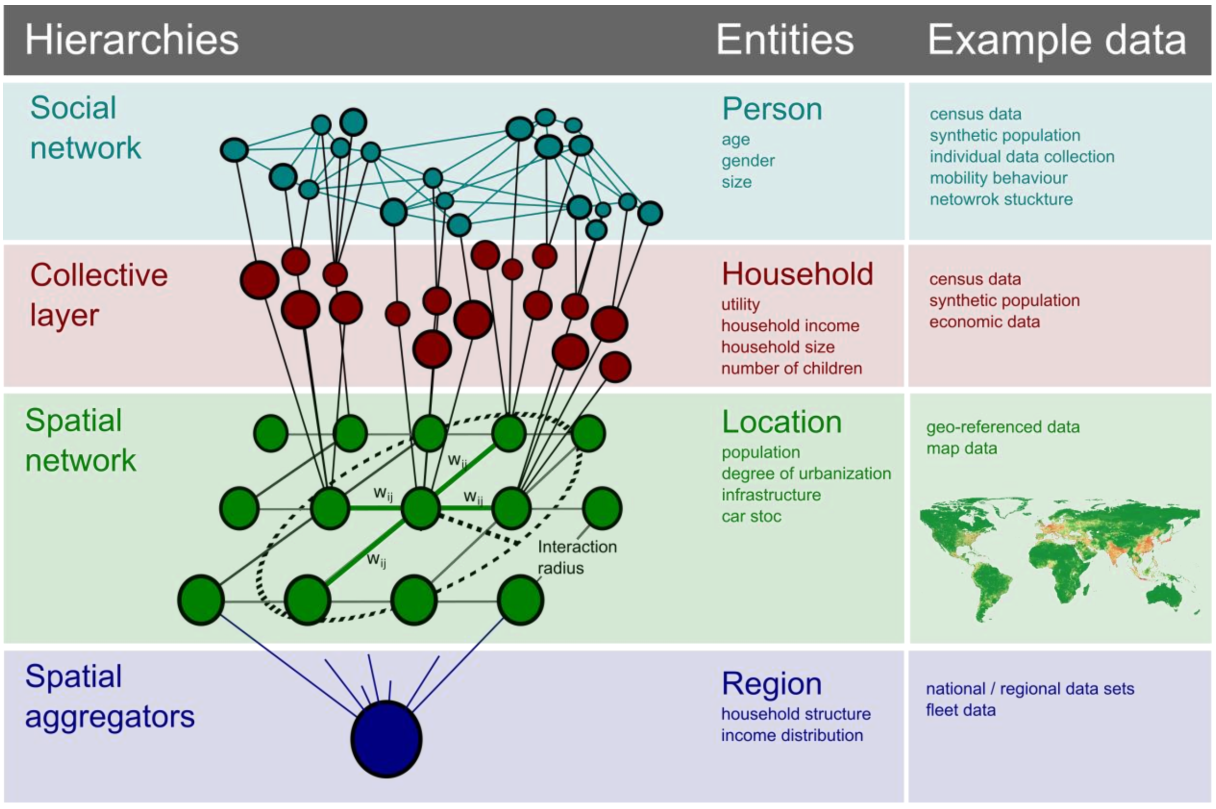

Figure 3: Example of the hierarchical structuring of interactions between agents in a graph.

3.5 Agent Traits

Currently, five traits for agents are implemented in ABM4py.

The `GridNode` trait assigns the agents type to be a part of a spatial domain. GridNodes or derived classes are required to construct a spatial domain using the subclass grid of the world. It also extends the register method of the agent in order to add the node in a dictionary storage that can be accessed by the x,y coordinate. This allows to access gridnodes not only be their ID, but also by their coordinate.

The trait `Parallel` is used in case of parallel model execution. It allows agents to acquire a globally unique ID and to organize on which process the original agent lives and where ghost copies exist.

The `Mobile` trait allows agents to move in the spatial domain. Putting this ability into a separated trait highlights a current limitation of the framework that mobile agents cannot be used in a parallelized model. Thus, it is currently not allowed to use the traits Parallel and Mobile simultaneously, which, however, will be changed in future versions.

The trait `Superpowers` is mainly added for completeness, but contradicts with the agent scope as described in section 3.1. In particular, it allows the agent to change the state of connected agent, which might be necessary for specific model purposes. Since applying to the agent scope is also the basis of the parallel execution, the Superpowers trait is also incompatible with Parallel trait at the moment.

3.6 Scheduling

A process overview can only be given for a specific model, but the topic of scheduling plays an important role in a description of ABM4py, due to potential parallel execution. Rubio-Campillo (2014) describes the scheduler as "the process that updates the set of agents and the environment that together form the model, and manages issues like the order of execution of the agents, and the way they interact with the other components (i.e., other agents, layers of raster maps, etc.)". In NetLogo, the observer plays this role: "it gives instructions to the other agents" . In ABM4py, scheduling is (mostly, we will get back to this) handled by the "world/ scheduler", that provides the functionalities for simulation setup and runs, as well as for the implementation of the model functions.

Scheduling is mainly the task of the modeler, only the parallel execution and some implementation details limit particular scheduling concepts. In ABM4py, time is discrete and only the current state of an agent is stored.

Therefore, the intuitive way to run a model is by executing one step after the other, however other (e.g. random or conditional) execution paths are possible. For the model part (the f above), the scheduler/world provides functionality for implementing the model dynamics, e.g. by making available dictionaries/lists/iterators, that can be used in defining "who does what in which order" .

In case of working with discrete step functions, each agent type might have a step that models its micro dynamics and an environment step organizes how the different agent steps are called. In ABMs, it is often useful to have an entity give instructions to a group of other entities, e.g., a household may trigger a change of state in all its members. This corresponds to smaller scheduling elements embedded in the model dynamics. However, an entity "giving instructions" to another one, so that this changes its state, may cause problems, if this second entity is at runtime actually computed by another process than the one that gives the instruction. Thus, it should be noted that smaller parts of scheduling are done by the agents and also can intervene with the concept of the agent scope. From a scheduling perspective, an agent (e.g. a household) can iterate over connected agents (e.g. persons) and therefore also overpower its scope. Thus, its remains the responsibility of the user since it cannot be restricted fully.

In case of parallelization, some care needs to be taken here, which will be discussed in section 3.7.

The main functionalities for scheduling that are provided by ABM4py are:

- Iteration over a particular agent type, a specified agent list, or agents that apply to a given filter rule
- Iteration over connected agents from the agent perspective
- Iteration over shuffled lists
- Random selection of agents of specific type

- Global and conditional access to agents attributes

If one considers an ABM as a dynamical system with state space X and transition function $f: X \rightarrow X$, the model can be described by providing this f, while a simulation applies f to an initial $x \in X$ and repeats this process for a specified number of steps. At each step, results computed from the state (which may also be selected parts of the state itself) that are of interest will be recorded, i.e., written to a file. For simplicity, of notation and thought, we consider an ABM a deterministic dynamical system. If it is not, which is mostly the case, we can think of the random number generator as included in the model. It is initialized with a seed and with each update of its state, it provides a new random number to the rest of the model.

Simulation setup functionalities further include reading additional geographic information files, reading synthetic populations , and creating collectives and agents. If the modeler does not define a path for output files, this is created with default names.

3.7 Parallelism

MPI (Message Passing Interface) is a standard API for communication between several processes in parallel computations. When starting a parallel program, several MPI-processes run the same program code using different parts of the data. Each process has an identifier called its rank, a group of MPI processes is called a communicator, and there is a predefined communicator MPI_COMM_WORLD that contains all processes. MPI-processes have independent memories and therefore need exchange data via the communicator. Without going into details, global communication includes “broadcasting” data to all processes, “scattering” data, so that each process gets its piece of the data, “gathering” data from all processes (to one or to all processes), and “reducing”, that is gathering with applying reduction operations to the gathered elements (again, in one or in all processes). Point-to-point communication is exchanging data between a sender and a receiver, however also different modes for this transfer are available. Sometimes, communication needs to be completed before a next computation is allowed: this can be assured by “wait” or “barrier” commands, where the latter tells all processes in a communicator to wait for all processes to reach this barrier.

If a model is run in parallel mode, ABM4py uses MPI and, as already briefly mentioned, partitions the spatial layer. The partitioning is currently not included in the framework, since it can be seen as a pre-processing step, that does not need to be to included in each model run, but executed one time. In addition, available software like metis, parMetis and others are well suited for this task and ABM4py be programed to interface with these. This means, at the beginning of a simulation, a model run is distributed to a number of processes that each compute the model dynamics for a set of locations on a given map of ranks. It is important to notice that the spatial domain needs to be partitioned in such a way as to balance the amount of work each processes carry out; For example, for a population of agents, the domain partitioning needs to balance the number of agents in each domain, not the number of location or grid nodes.

In order to allow agents to acquire information from connected entities computed by another process, the common concept of constructing ghosts is used. A ghost agent is a copy that replicates an agent's state on a process that needs this information but does not itself compute it. At certain (user-defined) time in a simulation, communication is needed to update ghost entities to keep track with the changes their originals have carried out in the meantime. ABM4py provides methods to initialize a distributed spatial domain and the automatic generation of ghost agents. Furthermore, based on this decomposed domain, creating an agent within areas of communication will automatically queue the necessary ghost copies, that can be than transferred at later stage of the modelers choice. Thus, the transfer and update of ghost agents is facilitated by predefined functions (see section 4), but controlled by the user.

An example of this domain decomposition and the concept of ghost agents is shown in Figure 4.

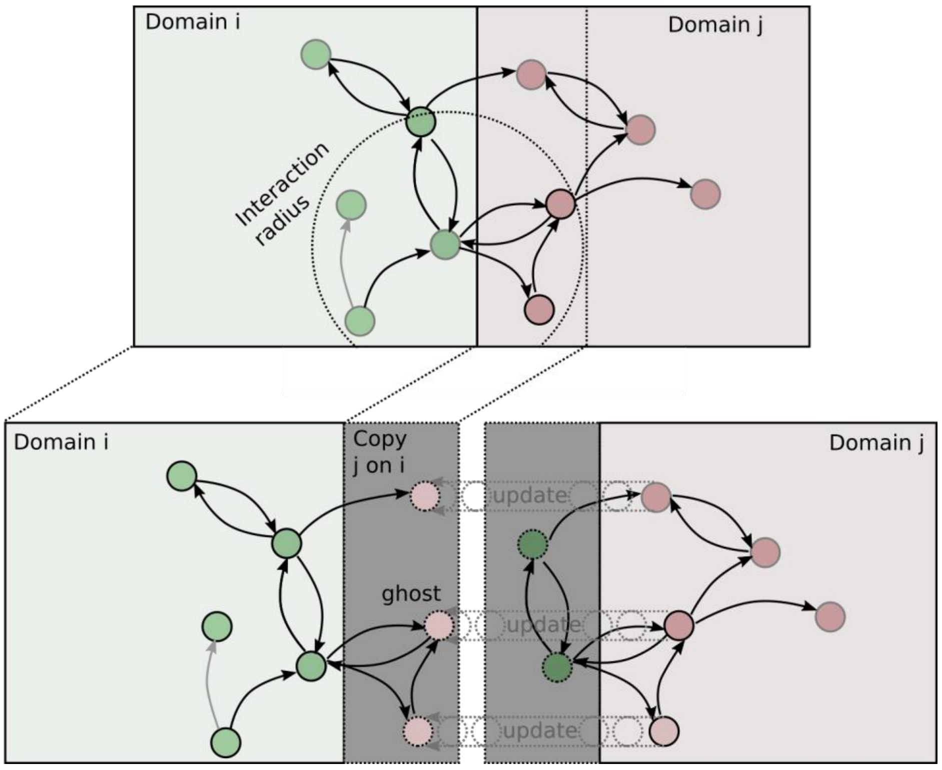

Figure 4 : Domain decomposition concept and illustration of ghost agents.

Naturally, this ghost concept comes with the disadvantage that before the update, the ghost might have a different state as the original. The modeler could only avoid that by updating after each action take by an agent that possesses ghost copies. This would lead to an communication overload and severe interconnection and waiting rules and is therefore infeasible. Thus, most of the time, other sources of uncertainty and approximations anyway pose larger restrictions and this ghost delay is often an acceptable problem.

4 Python implementation

The following section describes some python related implementation details that are helpful for users to familiarize with.

4.1 Agent

Agents are the core entity in ABM4py, and they are implemented as instances of classes, while each agent type refers to a different class, which can be derived from the given base classes and be extended by new functionalities. Multiple examples can be found in the example folder to describe the use of base classes in combination with traits.

Each agent can be interpreted as a combination of actions and an individual state. Often the abilities of agents of a single type are the same, but each agent differs in its state, which then leads to different actions. The state of each agent can be accessed by *self.attr*, which is actually a numpy view (pointer or memory reference) on a specific row in the data storage of the graph and can be used like a dictionary. This allows, both, a direct access from an agent viewpoint, but at the same time a centralized and efficient data structure. Of course the modeler has any freedom to save additional attributes in the agent class, however, only registered attributes that are stored in the graph interact with the framework and are accessible by methods provided by the environment.

Each agent is identified by an ID, which can be either accessed by *self.ID*, or *self.attr['ID']*. In case of parallel execution, *self.gID* refers to the global ID, which is unique over all processes and is used for communication and exchange of agents. The attribute *mpiOwner* references the current process of the agent, whereas *mpiGhostRanks* is a list of process IDs with ghost copies of the particular agent.

4.2 World

As already mentioned, World is the basic interface to access all other parts of the framework.

In general `get*`-methods provide the required access to agents, variables, parameters or sub-classes.

`register*`-methods are used to register new agent or link types, the underlying spatial grid, records for logging or individual agents.

`getAgentsBy*`-methods allow to access groups of agents based on different filter or grouping patterns.

`count*`-methods allow to count agents and links based on different filters or grouping patterns.

`getAttrOf*` and `setAttrOf*`-methods allow direct a global access the states of all agents and links. This access is always pointing to agents of a single type. Thus, actually the agent type is therefore only derived from the first agent ID in a given list.

`glob2loc` and `loc2glob` relate to the conversion between global and local IDs

`finalize` closes all hdf5 output files and stores records and given parameters.

4.3 Graph

As mentioned before, the underlying graph structure is a self-implemented directed graph. The main data structure is the "structured array" from the module numpy, which is used to save all state variables of all links and nodes. Since different types have different state variables, all types are stored in individual arrays.

After the registration of a new type (`world.registerNodeType`), the node array is initialized with an initial size. At the runtime of the registration, the state variables need to be defined, by their name and optionally type and length in case of vectors. In addition, dynamic and static state variables are distinguished, because static variables are saved only once. If this size is being exceeded by adding more agents/links, the size is dynamically increased by a factor of two, or the required new size in case of the creation of many nodes/edges.

Storing links requires additional attributes like sourceID and targetID, but currently no global IDs. In addition, to allow efficient access to links, expensive search operations need to be avoided. Thus, links and their connected nodes are stored in multiples way which creates an overhead for creation and the deleten of links, but improves access significantly.

For each node type, the following relations are stored:

- (sourceID, targetID) -> edgeID
- nodeID -> edgeIDs of outgoing and incoming edges
- nodeID -> nodeIDs of nodes connected by outgoing or incoming edges

Thus, methods, e.g. for accessing of connected nodes or edge attributes (`agent.getPeerAttr`), check the connectivity between agents (`world.areConnected`) can act on the respective efficient storage type for the relation.

4.4 Parallel agent passing interface (PAPI)

The core of the all parallelization efforts is the python package mpi4py which provides the python interface to the underlying MPI API. Most of the known and basic functions of MPI are transferred to mpi4py. However, the current implementation mostly relies on all2all communication to transfer distributed variables and agent states. This allows the MPI library to optimize the communications streams, however might become inefficient for number of processes beyond a hundred.

At the current state, PAPI offers only one limited way to initialize the exchange of agents between processes. For a specific type of agents, each process needs to generate the ghost agents that are expected to exist on the process. This implies that all processes know the entire domain and the origin of the ghost agents. In the current process, the init function of the grid class does this automatically based on an array of MPI ranks. In this spatial case, the ghost locations define the interaction radius of the grid nodes and all dependent agent types that are connected with the grid nodes.

After that, the method `world.papi.initAgentCommunication` initializes the communication between processes, the exchanges the first ghost agents and updates their initial state. From this moment on, the originals of all processes have knowledge of possible ghost copies on distributed processes. This is important since with this information the creation of agents triggers the automated creation of a ghost copy. The rationale behind the creation is that each agent connected to the grid node is within the interaction radius of agents on process with ghost copies of this grid node. Thus, the newly connected agent will be transferred to the other process to allow the potential interaction within the defined radius. Since sending on ghost copy at a time is highly inefficient, the modeler need to manually start the sending process using `world.papi.transferGhostAgents`. For the same reason, the method `world.papi.updateGhostAgents` needs to be called for updating the state of ghost agents, which can act on all agent types and state variables or respective subsets.

Furthermore, mpi4py distinguishes two modes of communication, one is based on numpy data types and the other relies on the module *pickle* to send objects of arbitrary types. Currently, only the *pickle* mode is used in ABM4py, even though the numpy based approach is claimed to be several magnitudes faster for bigger message sizes. Thus, the transfer of ghost agent and their states via the numpy mode is planned for the future .

4.5 IO

For input/output (IO) operations, ABM4py relies mostly on the modul *h5py* with works quite well in combination with numpy and mpi4py. It supports structured arrays from numpy and can use mpi4py for parallel IO. Thus, the functions `initAgentFile`, `writeAgentDataToFile` and `finalizeAgentFile` can be used to control the output of agent state variables and respective functions for links are provides as well.

In addition, `writeAdjFile` can be used to create an adjacency file for a subgraph of a specified node type. This is important as an interface to partitioning software like for example metis.

5 Scaling

One final word about the scalability of this framework need to be mentioned, since this is always of utter importance in the HPC community. The main reason is to avoid running inefficient code on big computing clusters and wasting this very expensive computing power.

Even though we emphasize that this is a prototyping framework and the scalability will the code will always mainly depend on the actual model implementation, it is important to show that the framework allows for a certain quality of scaling. Therefore, this section provides a scalability test results which was done using the Green Growth Pilot model. A detailed description of the model can be found in previous deliverables from Work package 4. Here we will present a short summary of technical specifications.

The model consists of about 26,000 grid agents that represent the spatial extent of Germany and is populated with about in 3.3 million agents in roughly 1.75 million households. All of them are explicit agents instance. The grid agents cover the extend of 5x5 kilometers and the interaction radius is set to 35 km (7 grid nodes). This lead to an overall number of about 30 million spatial connections and roughly 300 million connections in the social network between agents.

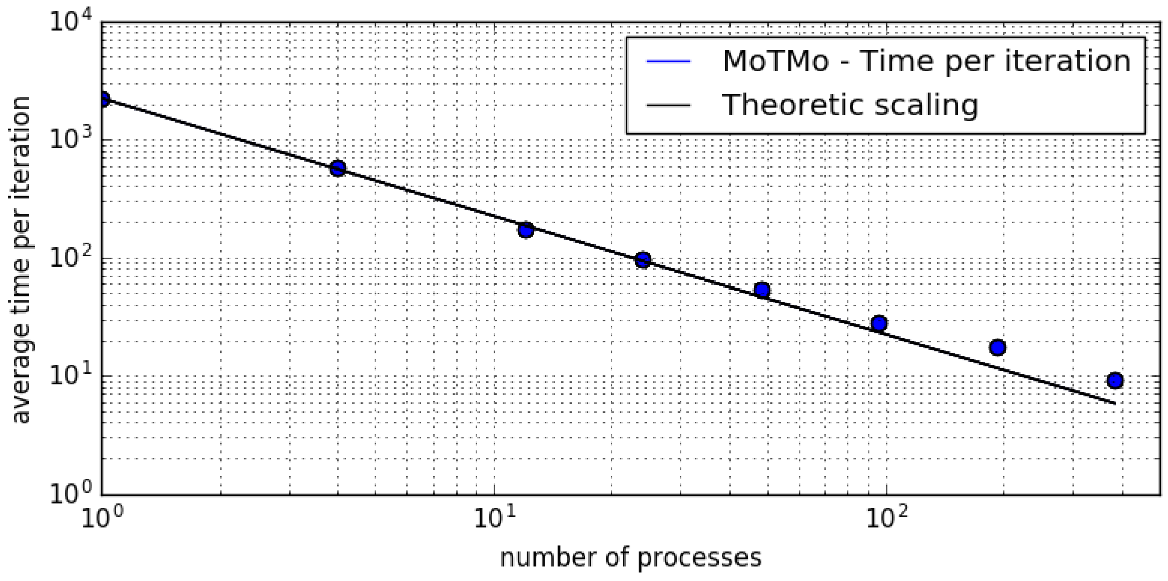

Figure 5: Exemplary results for a scaling test using the model MoTMo of the Green Growth Pilot. The times shown on the y-axis are the times per single model step.

Figure 5 shows the resulting average iteration times for a single step and highlights the linear reduction of the computing time with the number of processes. Table 1 provides the corresponding numbers and some additional information.

number of processes	ration of ghost agents	time for simulation init	time per iteration	Strong scaling efficiency (init)	Strong scaling efficiency (iteration)

1	0	24976 s	2244.5 s	1	1
4	0.09	7872 s	582.7 s	0.79	0.95
12	0.81	3125 s	171 s	0.67	1.09
24	1.70	2110 s	97.27 s	0.49	0.96
48	1.50	1888 s	53.07 s	0.28	0.88
96	3.07	922 s	27.9 s	0.28	0.83
192	5.86	308 s	17.80 s	0.42	0.65
384	10.43	189	9.11 s	0.34	0.64

Table 1: This tables summarizes the scaling of the MoTMo model of the Green Growth Pilot.

The ratio of ghost agents is the ratio of created ghost copies and actual agents on the different processes. It is provided, since for high number of processes, the fixed interaction radius lead to strongly increased ratio of ghost agents, which prohibits further speed up possibilities. In the last run with 384 processes, the spatial partitioning in combination with the fixes interaction radius leads to roughly ten times more ghost copies on a process than real agents. This easily explains the drop in the efficiency above 100 processes and therefore, for a bigger example we can still expect better scaling performance.

The time for the simulation initialization contains serial code parts and the loading of data, therefore the init times scale less well.

6 Documentation, design concepts and further steps

Currently the documentation of individual classes and functions can be found in the doc folder. In addition, examples and tutorials are provided and cover the most basic simulation workflows. However, ABM4py is still under construction, thus we aim to add more extensive material in future releases.

The ODD protocol provides a structure for describing individual- or agent-based models that has been widely accepted. While here we do not describe an ABM but a tool for implementing these, the general structure of ODD, and some of its elements are helpful to draw on also for this task. We will therefore first present an overview, that focuses on ODD's elements "purpose", "entities", as well as "process overview and scheduling". ODD's design concepts are only very briefly touched upon, but then the details required to document ABM4py for the inclined user are presented.

ODD's design concepts are naturally model specific and hence beyond the scope of this text (especially so for the concepts "basic principles" and "emergence"). However, we wanted to very briefly mention that, ABM4py provides functionality for the users to include collectives in their models, and that, as an outlook, further development of ABM4py could provide different functionalities in view of some of the other design concepts. Templates could be provided for simple implementation of different kinds of adaptation, objectives, learning, prediction, sensing, interaction, stochasticity, and observations. As an example, objectives could contain utility maximization: here, the user could fill a specific utility function and an optimization algorithm into a template.

The current version of ABM4py is considered under development and some important features are still missing. Thus, it will improved concerning the features, performance and user interface. However, the generality will rely on a future community of users that consider ABM4py a useful tool for their work and research.

References:

Epstein, Joshua M.: "Agent-based computational models and generative social science." Complexity 4.5 (1999): 41-60., 1999

Dum, R and Johnson, J: "Global Systems Science and Policy", *Non-Equilibrium Social Science and Policy*, Springer, 209–225, Eds: Johnson, Jeffrey,Nowak, Andrzej, Ormerod, Paul, Rosewell, Bridget, Zhang, Yi-Cheng, 2017

Fuerst S. and Geiges, A.: "Distributed agent graph", GCF-Working paper, 2018, https://globalclimateforum.org/2018/09/26/gcf-working-paper-03-2018-distributed-agent-graph

Grimm V, Berger U, De Angelis DL, Polhill JG, Giske J, Railsback SF: The ODD protocol: a review and first update. Ecological Modelling 221:2760–2768, 2010

North, M.J., N.T. Collier, J. Ozik, E. Tatara, M. Altaweel, C.M. Macal, M. Bragen, and P. Sydelko, "Complex Adaptive Systems Modeling with Repast Simphony, " Complex Adaptive Systems Modeling, Springer, Heidelberg, FRG, 2013

Rubio-Campillo: "Pandora: A Versatile Agent-Based Modelling Platform for Social Simulation." ,SIMUL 2014: The Sixth International Conference on Advances in System Simulation, 2014