Matthias Scheutz and Paul Schermerhorn

Artificial Intelligence and Robotics Laboratory

Department of Computer Science and Engineering

University of Notre Dame

Notre Dame, IN 46556

Adaptive Algorithms for the Dynamic Distribution and Parallel Execution of Agent-Based Models

Abstract

We propose a framework for defining agent-based models (ABMs) and two algorithms for the automatic parallelization of agent-based models, a general version **P-ABM**$_G$ for all ABMs definable in the framework and a more specific variant **P-ABM**$_S$ for "spatial ABMs" targeted at SWARM and ANT-based models, where the additional spatial information can be utilized to obtain performance improvements. Both algorithms can automatically distribute ABMs over multiple CPUs and dynamically adjust the degree of parallelization based on available computational resources throughout the simulation runs. We also describe a first implementation of **P-ABM**$_S$ in our SWAGES environment and report both results from simulations with simple SWARM agents that provide a lower bound for the performance gains achievable by the algorithm and results from simulations with more complex deliberative agents, which need to synchronize their state after each update cycle. Even in the latter case, we show that in some conditions the algorithm is able to achieve close-to-maximum performance gains.

Key words: agent-based model, adaptive parallelization and distribution

PACS:

---

1 Introduction

Agent-based models (ABMs) and their simulations have been widely employed in the fields of complex systems, artificial life, genetic programming and genetic evolution, social studies, and others to study emergent group behaviors in swarms [1–5], bacterial chemotaxis signaling pathways [6,7], population ecology [8–10], social and economic systems [11–13], and many more. [1]

While most ABMs are intrinsically parallel in that they implicitly decompose the overall complex system behavior into tractable behaviors of individuals and their interactions, most simulation environments for ABMs are sequential: they run on a single CPU and do not support the distribution of the ABM over multiple CPUs (even though they might support the scheduling of sequences of ABM simulations). And the few existing simulation environments that support the parallelization of ABMs, require the designer of the ABM to make provisions in the code that will allow for the distribution of the simulation. As a result, it is very difficult for non-experts (i.e., scientists with no background in parallel programming) to utilize the potential parallelism present in many ABMs for running parts of their models in parallel. Moreover, such parallelizations are typically not generic, but are tailored

Email address: {`mscheutz,pscherm1`}`@nd.edu` (Matthias Scheutz and Paul Schermerhorn).

[1] Agent-based models–sometimes also called "individual-based" models–are often used to simulate the behavior of complex real-world systems, when possible state changes of individual entities are known and can be encoded in rules, while no such knowledge exists for global world states (e.g., the state given by the environment and all its agents).

to a specific execution environment (e.g., a given number of hosts, CPUs, etc.) and thus require modifications for simulation runs in different execution environments.

We believe that an algorithm that can *automatically* and *dynamically* distribute a given agent-based model over a dynamically changing set of CPUs without requiring any assistance from the user would be of great utility for the ABM community; most importantly, because the same ABM simulation will work both in sequential and parallel computing environments, but also, because it will be possible to maximally exploit the available computational resources at any given time.

In this paper, we introduce a formal framework for capturing agent-based models and two subsequent algorithms for the automatic parallelization of agent-based simulations defined in the framework, a general version **P-ABM**$_G$ for all agent-based models and a more specific variant **P-ABM**$_S$ for "spatial ABMS". The spatial variant is especially targeted at the SWARM and ANT-based models, which typically are defined in the context of metric spatial environments, where the additional spatial information can be utilized to obtain performance improvements. Both algorithms can automatically distribute ABMs over multiple CPUs and dynamically adjust the degree of parallelization based on available computational resources throughout the simulation runs. We also describe a first implementation of **P-ABM**$_S$ in our SWAGES environment and report results from simulations with simple SWARM agents and more complex deliberative agents that use a version of the A^*_ε algorithm [14] for planning routes in a dynamic environment. The former provide a lower bound of the performance gains achievable by the algorithm, the latter demonstrate that the algorithm is also useful for complex agents that require complex computations to compute their behaviors.

2 A Description Framework for Agent-Based Models

Agent-based modeling has been applied in many diverse fields (as mentioned in the introduction). Consequently, different kinds of formalisms and frameworks have been developed to capture this diversity (e.g., some models are essentially physics-based, while others operate on a mere social level). Common to all of them is the idea that individuals (in groups of entities) are modeled as such (e.g., molecules in a cell, ants in a colony, strategies in a tournament, etc.), rather than as relations among global state variables abstracting over the individuals (although the latter typically can be recovered from the individual's interactions). Hence, simulations of these models could all potentially benefit from parallelizing the model at its natural seams: the individuals. Specifically, it would be beneficial for ABM research if ABM simulations could be distributed over a set of CPUs such that the overall execution time of a simulation run can be reduced, ideally by a linear factor in the number of employed CPUs. Moreover, it would be particularly helpful if ABM researchers did not have to worry about the parallelization themselves, but rather could defer it to simulation software, which adaptively distributes an ABM over CPUs as they become available. In this paper, we will propose algorithms that can accomplish this task.

A prerequisite for the definition of any such algorithm is a formal description of the notion of "agent-based model", for otherwise it is only possible to define ad-hoc algorithms for specific model instances. Clearly, the generality of the description will limit the applicability of the algorithm. Hence, it is important to take care in laying out the formal description framework to be as inclusive as possible, while at the same time not sacrificing the potential of the algorithm to save time (via parallelization). We will attempt this balancing act in the rest of this section by

first introducing a general framework for ABMs that is intended to capture what is essential about ABMs, and then extending the notion of "agent-based model" to "spatial agent-based model", a class of models that is particularly amenable to parallelization. [2] The proposed framework is based on a formalism introduced to define multi-level, multi-scale agent-based models [15].

2.1 Agent-Based Models

ABMs are generally models of some behavior of (real-world) entities in their environment over a period of time. "Agents"–in the context of ABMs–are then the representations of these *real-world entities*, whose behavior is characterized by a set of *rules* that determine the state change of an entity based on the past state of the entity and (possibly) environmental states. These rules are encoded in the representing "agents" within the ABM. A *simulation of an ABM* is a computational process that starts in *some initial condition* and then updates the agents' states and environmental states over time, thus providing a model of the evolution of the real-world system (within the confines of the rules and state representations chosen in the ABM to model the real-world system).

A subset of agent-based models (e.g., SWARM or ANT models), very common in *artificial life simulations* or socio-economic models, includes spatial assumptions about the environment. [3] Typically, the environment is modeled as a discrete or

[2] We believe that most ABMs in the literature can be translated into the proposed formalism, but for space reasons we cannot give translation for a representative set of models here.

[3] Examples of non-spatial agent-based models are versions of the iterated prisoner's dilemma, where all participating strategies are modeled as agents playing all other agents

continuous *metric* space (e.g., with the Euclidean norm). Such models allow for the simulation of interactions among entities based on a notion of *distance*, which is crucial for understanding the behavior of many biological systems and organizations of agents in physical spaces (e.g., insect swarms, flocks of birds, schools of fish, etc.). In a sense, any agent-based model can be viewed as a "spatial agent-based model" (S-ABM)–if no notion of space is defined in an ABM M, it can be trivially viewed as an S-ABM with only one location, which all entities occupy. Hence, we start with a general characterization of (metric) S-ABMs, including the main constituents: (1) a spatial environment, (2) a set of global (or environmental) states, (3) a set of locations in the environment for which global states can be updated, (4) a set of entity types (of all entities defined in the model), (5) a set of initial conditions (some/all of which are to be investigated in simulation runs of the model), and (5) a set of conditions that single out final configurations (i.e., configurations that mark important states of the model, typically used for terminating simulations, such as equilibrium states or states without any entities left, etc.). [4]

Definition 1 (Spatial Agent-Based Model) *A* spatial agent-based model $\mathcal{M} = \langle Env_{\mathcal{M}}, GS_{\mathcal{M}}, Loc_{\mathcal{M}}, Ent_{\mathcal{M}}, Init_{\mathcal{M}}, Cond_{\mathcal{M}} \rangle$ *consists of an n-dimensional bounded or unbounded metric space* $Env_{\mathcal{M}}$*, a set of global environmental states* $GS_{\mathcal{M}}$*, a set of locations in the environment together with their respective update functions* $Loc_{\mathcal{M}} = \{\langle L, U_L \rangle | L \in \mathcal{P}(Env_{\mathcal{M}}, U_L : \mathcal{P}(Env_{\mathcal{M}} \times GS_{\mathcal{M}} \mapsto GS_{\mathcal{M}})\}$*,* [5] *a set of entity types* $Ent_{\mathcal{M}}$ [6]*, a set of initial configurations* $Init_{\mathcal{M}}$*, and a set of conditions for final*

in every round of a tournament.

[4] The qualifier "metric" was chosen simply because we are not aware of any non-metric spatial ABMs, but nothing theoretical hinges on it. We will focus on n-dimensional continuous Euclidean spaces (discrete Euclidean spaces are then just a special case).

[5] We use "$\mathcal{P}(X)$" to denote the power set of a set X.

[6] We prefer the term "entity" to refer to the computational representation of some real-

configurations $Cond_{\mathcal{M}}$. Typically, $Cond_{\mathcal{M}}$ will consist of logial formulas (e.g., in first order logic) that describe properties of configurations.[7] *Each configuration in the set of initial configurations is $\mathcal{C} = \{E|\tau(E) \in Ent_{\mathcal{M}}\} \cup first\ Loc_{\mathcal{M}}$, i.e., a set of instantiated entities E (whose types $\tau(E)$ are in $Ent_{\mathcal{M}}$ together with all the defined locations, i..e, the first projections of $Loc_{\mathcal{M}}$).*[8]

We will use $Cfg_{\mathcal{M}}$ to denote the set of all possible configurations determined by the model $\mathcal{M}$. If we assume discrete environments with finitely many locations and entity types with only finitely many instances, then the set of configurations will also be finite. Note that locations have their own update functions directly associated to allow for local and global changes of environmental states (e.g., global and/or local temperature, updates pheromone concentration, etc.).[9]

---

world entity within an ABM over the more common term "agent" given the ambiguities in the literature associated with the term "agent", which we would like to avoid (e.g., any computational entity within an agent-based simulation is called "agent" even though it might not even have sensors or actuators, which are requirements for many notions of agent, e.g., as in an "autonomous computational entity with sensors and actuators that is situated in some environment").

[7] The reason for including conditions for final configurations instead of a set of final configurations in the definition of the model is that the former is more intuitive, closer to practical applications of ABMs and directly implementable, while the latter might obscure what is common to these configurations and might require the specifications of general conditions that define the set of final configurations to be efficiently implementable after all.

[8] Note that by "instantiated entity E" we always intend the state of an entity of type E. Also, we use "FIRST" to denote the function that returns the first component of a typle.

[9] For S-ABMs where new locations can appear and old ones can disappear at various times a different way of introducing locations (e.g., as functions of environmental states over time or as entities themselves as we have done in the past) might be formally more succinct.

2.2 Entities in Spatial Agent-Based Models

There are many ways in which entities in S-ABMs could be formally characterized, each with its own advantages and disadvantages (e.g., some are easily accessible for the non-expert programmer, others are quite technical and focus on a mathematically precise formalization in set theory). We chose to formalize entities in ABMs as consisting of a *body* (i.e., a representation of their physical extension and shape in space) together with *sensors* that can measure the environment and *actuators* that can affect it. Hence, interactions between entities can only occur via sensors and effectors. This includes communications among agents, which in some simulation environments can take place directly (e.g., analogous to "telepathy"). [10] The advantage of our proposed approach is that communication is treated as a special case of sensing and acting (just as in the real world), and thus does not require any additional, special formal apparatus (nor any special provision in the parallel algorithms to be proposed later).

The conceptual separation between body, on the one hand, and sensors and actuators, on the other, is intended to make it easier to model entities that share the same geometric shape and bodily characteristics (e.g., in terms of the extension in space), but differ with respect to their sensory and actuator capabilities (e.g., sensory and actuator range, modalities, etc.). With this separation it is possible to define rules that govern the behavior of the sensors independently of those used for the body. This is, for example, useful for investigations of the effects of sensory range on agent performance (e.g., [16]).

[10] One consequence of our model is that communication ranges are determined by sensory and actuator ranges. If unlimited communication is required, then the corresponding ranges need to be set accordingly, e.g., see [16].

Definition 2 (Entity, Body, Sensor, Actuator) *An* entity $E = \langle B_E, C_E, U_E \rangle$ *consists of a body* B_E*, and possibly a controller* C_E*, and an update function* U_E *that updates the state of the entity (see below).*

A body $B = \langle Sen_B, Geom_B, Loc_B, Int_B, Act_B, U_B \rangle$ *consists of a sequence of sensors* Sen_B*, a geometry configuration* $Geom_B$ *of its shape (including where sensors and actuators are located), a location* Loc_B *where the body is situated, a sequence of internal bodily state variables* Int_B *(e.g., for energy sources), a sequence of actuators* Act_B *and a mapping* U_B *from a sequence of sensor states, geometry states, locations, and a sequence of internal states to a sequence of actuator states, geometry states, locations, and a sequence of internal states (e.g.,* U_B *could be given by a set of differential equations or difference equations).*

A sensor $S = \langle Loc_S, Rng_S, Out_S, T_S \rangle$ *consists of a location of the sensor* Loc_S *in* $Env_{\mathcal{M}}$*, a sensory range* $Rng_S \subseteq Env_{\mathcal{M}}$*, a set of output states* Out_S *and a transformation function* $T_S : first\ Loc_{\mathcal{M}} \times Cfg_{\mathcal{M}} \mapsto Out_S$ *that maps sensor locations with respect to any arrangements of entities to sensor output states.*

An actuator $A = \langle Loc_A, Rng_A, In_A, T_A \rangle$ *consists of a location* Loc_A *of the actuator in* $Env_{\mathcal{M}}$*, an actuator range* $Rng_A \subseteq Env_{\mathcal{M}}$*, a set of input states* In_A *and a transformation function* $T_A : first\ Loc_{\mathcal{M}} \times In_A \times Cfg_{\mathcal{M}} \mapsto first\ Loc_{\mathcal{M}} \times Cfg_{\mathcal{M}}$ *that maps actuator locations based on actuator input and the given configuration onto a new configuration and new actuator location (e.g., if the actuator is attached to a body, this will allow for moving the body).*

Define the sequences of sensor and actuator states $In_B(i) := Out_{S_i}$ and $Out_B(j) := In_{A_j}$, where $S_i = Sen_B(i)$ (i.e., the i-th sensor in the body B) and $A_j = Act_B(j)$ (the j-th actuator in the body B), respectively. Based on these definitions we can now formally define the bodily update function $U_B : In_B \times Geom_B \times Int_B \times Loc_B \mapsto$

$Out_B \times Geom_B \times Int_B \times Loc_B$. A triple $\langle g, i, l \rangle \in Geom_B \times Int_B \times Loc_B$ (without input and output states) is called *bodily state of the entity*.

The above definition of the "body" of an entity E is sufficient to describe autonomous entities in the sense of "agents" in many agent-based simulations. Hence, an entity E can consist of only a body B_E (as mentioned above), in which case $U_E = U_{B_E}$. However, we believe that it is often useful to separate bodily processes from control processes (e.g., to allow for the possibility of having different controllers for the same kind of body). Hence, we allow entities to include a separate controller whose whole purpose is to control the entity's body based on its bodily states. [11]

Definition 3 (Controller) *A* controller $C = \langle In_C, Comp_C, Out_C, U_C \rangle$ *has a sequence of inputs* $In_C : In_{B_E} \times Int_{B_E}$*, a sequence of outputs* $Out_C : Out_{B_E} \times Int_{B_E}$*, a sequence of internal computational states* $Comp_C$ *and a mapping* $U_C : In_C \times Comp_C \mapsto Comp_C \times Out_C$ *from input and internal states to output and internal states (e.g., the mapping could be specified via condition-action rules or feedback equations from control theory).*

Entity update functions U_E for entities E that consist of both a body B_E *and* a controller C_E have to be defined both in terms of the body and the controller update functions $U_E : In_{B_E} \times Geom_{B_E} \times Int_{B_E} \times Loc_{B_E} \times Comp_{C_E} \mapsto Out_{B_E} \times Geom_{B_E} \times Int_{B_E} \times Loc_{B_E} \times Comp_{C_E}$. The *state of an entity* is then given by its bodily and control states, i.e., the quadruple $\langle g, i, l, c \rangle \in Geom_{B_E} \times Int_{B_E} \times Loc_{B_E} \times Comp_{C_E}$. The bodily output ($Out_{B_E}$) is sent to the actuators Act_B, which attempt (but potentially fail) to carry out the actions specified. This last point is important in modeling sit-

[11] Note that the above definitions do not allow for "disembodied controllers" as we believe that this concept cannot be coherently formulated in the present context.

uations, where an entity's controller might attempt to achieve a state that is inconsistent with the environmental state (e.g., the control attempts to move the entity forward even though an obstacle is blocking the way). In such a case, the bodily actuator state has precedence over the controller commands (e.g., in a physics simulation the actuator transition function will always obey the laws of physics, regardless of attempts by the controller to produce forces that are not possible in the given configuration). A controller, and consequently the entity of which it is a part, can thus only *attempt* to affect the environment without any guarantees that "motor commands" can and will actually be executed.

2.3 Simulations of Agent-Based Models

In the above exposition, we introduced all individual U_E as functions; moreover all locations also had update functions associated with them. Hence, each configuration C_i has a unique successor configuration C_{i+1} and we can define an overall "model update function" $U_{\mathcal{M}}$ on the set of configurations in terms of the update functions of each entity and each location:

Definition 4 (Successor Configuration, Model Update Function) *Let* $C_i \in Cfg_{\mathcal{M}}$ *be a configuration in a spatial agent-based model* $\mathcal{M}$. *Then the* successor configuration of $C_i' = U_{\mathcal{M}}(C_i)$ *is given by* $U_{\mathcal{M}}(C_i) = \{U_E(E)|E \in C_i\} \cup \{U_L(L)|L \in first\ Loc_{\mathcal{M}} \wedge U_L \in first\ Loc_{\mathcal{M}}\}$. $U_{\mathcal{M}}$ *is called the* model update function *of the model* $\mathcal{M}$.

Given that U_M is a function, sequences of configurations–call them "simulations"–are entirely determined by the initial configuration C_0 of a model. This leads to the following definition:

Definition 5 (Simulation of an S-ABM) *A* simulation *of an S-ABM $\mathcal{M}$ is defined as a finite sequence of configurations $U^i_{\mathcal{M}}(C_0) = \langle C_0, C_1, C_2, \ldots, C_{final} \rangle$ starting with an* initial configuration $C_0 \in Init_{\mathcal{M}}$ *and ending in the* final configuration C_{final} *(as determined by the conditions in $Cond_{\mathcal{M}}$, i.e., $Cond_{\mathcal{M}}(C_{final}) \wedge \neg Cond_{\mathcal{M}}(C_{i \neq final})$), where all $C_i \in Cfg_{\mathcal{M}}$ ($0 \leq i \leq final$). A transition between two subsequent configurations C_i, C_{i+1} ($0 \leq i < final$) in a sequence is also called* simulation step. *The duration of the simulation is measured in terms of* simulation cycles *(or just* cycles, *for short), i.e., $|\langle C_0, C_1, C_2, \ldots, C_{final} \rangle|$, and we use the term "cycle" to refer to the position of configurations in the sequence of configurations.*

Simulations for ABMs, as defined, are consequently deterministic and reproducible from initial states. However, it is sometimes more convenient to allow for "non-deterministic" state transitions in models (both for entities and locations), e.g., to model probabilistic state transitions where each transition has a certain likelihood associated with it. In such a case, it is straightforward to augment the above definitions of entities and locations by dropping the requirement that entity and location updates be functional, but construct them as annotated relations, where the annotation is a numeric value in $[0,1]$–the transition probability–such that all annotations of transitions from a given state with the same update sum to 1. The consequences for implementations are that explicit representations of random number generators and their seeds are necessary to be able to reproduce simulations. Reproducible simulation runs are then defined in terms of the seeds of the random number generators and the initial states (i.e., at any choice point the random number generator will deterministically produce a next "random number", which is used to determine the state transition).

We are now interested in general algorithms $\mathcal{A}$ that compute simulations of agent-based models (defined in the proposed framework) in the following sense: for any

configuration $C \in Cfg_{\mathcal{M}}$ given as input, $\mathcal{A}$ produces the successor configuration as output $U_{\mathcal{M}}(C_i)$. In particular, we are interested in parallel algorithms that take initial configurations of agent-based models and return final configurations of simulations, where intermediate configurations are distributed across a possibly dynamically changing pool of multiple CPUs.

3 Parallel Simulation Algorithms

Spatial agent-based simulation models can be automatically parallelized and distributed in different ways. One obvious way is to run each entity on its own CPU. Before an entity can update its state, it needs to collect the current state information from all other entities (running on other CPUs). Once the information is available, the entity updates its state and begins the update cycle again. All CPUs update their respective entities in the very same cycle-based fashion to ensure the correctness of the results.

Another possibility is to determine in advance whether an entity needs the state of another entity for its update and to distribute agents based on these dependencies (e.g., subsets of mutually dependent entities end up on the same CPU, which limits the exchange of state information to local entities).

Other options are to predict or (empirically) determine the actual update time of an entity and run computationally expensive entities on separate CPUs, while running computationally cheap entities together on one CPU.

Which of the above approaches works best will depend on various factors, including the complexity of the update function of the involved entities, the distribution of entity types in a particular setup, the computational overhead of sending state in-

formation requests and receiving them, the pool of available CPUs (e.g., individual speeds, etc.) and whether it remains constant throughout a simulation run or can change over time, etc. All these factors (and their interdependencies) are important for efficient parallelizations of agent-based models. For space reasons, however, we will have to limit our discussion to two factors: the automatic distribution of entities over a pool of available homogeneous CPUs and the automatic, dynamic adjustment of this distribution based on the changes of that pool over time.

We will start with a discussion about splitting configurations, then give a formal characterization of the general algorithm and proceed to the special spatial variant. We will also show that both algorithms are correct in the sense that they will yield the same results as a sequential version.

3.1 Splitting Configurations

We start by formalizing the intuitive idea of splitting up a set of entities and assigning them to processors in a given set of CPUs.

Definition 6 (Split of Configuration) *Let $\mathcal{M}$ be a S-ABM, C a configuration in $Cfg_{\mathcal{M}}$, and $Proc = \{CPU_1, CPU_2, \ldots, CPU_n\}$ a set of available processors ("processor pool"). Then a* split P^C_{Proc} *of C is a mapping $P : C \mapsto Proc$–called* entity-processor assignment*–of entities to processors CPU_i in $Proc$.*

Note that the entity-processor assignment does not have to be surjective as we might not need all processors in the processor pool.

Corollary 7 *A split P^C_{Proc} induces a partitioning Π_C of a configuration C into i disjoint subsets of entities Π_{C_i} in C.*

PROOF. It is straightforward to check that the sets $\Pi_{C_i} := \{E | E \in C \wedge P^C_{Proc}(E) = CPU_i\}$ for each $CPU_i \in Rng(P^C_{Proc})$ are a partition of C (they are disjoint and their union is C).

Each Π_{C_i} is itself a configuration and can thus be updated in the same way as C. In the context of parallelizing a simulation, i.e., a sequence of configurations, we can simply split the initial configuration C_0 of a simulation among the processors in *Proc* and then continue to update the entities on each processor CPU_i independently as long as the states of entities updated by CPU_i do not depend on the states of entities updated by other processors CPU_j. If there is such a dependence, then there are two options: (1) either the external state information has to be obtained before the state of the local entities can be updated, or (2) both configurations are "merged" before the update (we will consider both options below).

Hence, the critical aspect in parallelizing a spatial agent-based simulation is to detect these dependencies automatically and communicate the necessary information among processors. We will first formally define the notion of "update independence", and then propose a necessary condition for detecting it.

Definition 8 (Update Independence) *An entity E_1 is* update-independent, $\mathtt{UI}_C(E_1, E_2)$, *of another entity E_2 in a configuration C (with $E_1, E_2 \in C$), if the updated state of E_1 in the successor configuration C' of C is the same as in the successor configuration $(C - E_2)'$ of $C - E_2$ (i.e., the configuration obtained from C by removing entity E_2). E_1 is called* update-dependent *on E_2 in C if $\neg\mathtt{UI}_C(E_1, E_2)$. E_1 and E_2 are called* mutually update-independent *in C if E_1 is update-independent of E_2 and vice versa (see Figure 1). Two subconfigurations $C_1, C_2 \subseteq C$ of C are* mutually update-independent $\mathtt{UI}_\mathtt{C}(\mathtt{C}_1, \mathtt{C}_2)$ *if* $\forall E_1 \in C_1, E_2 \in C_2 [\mathtt{UI}_{C_1}(E_1, E_2) \wedge \mathtt{UI}_{C_2}(E_2, E_1)]$. *A set of*

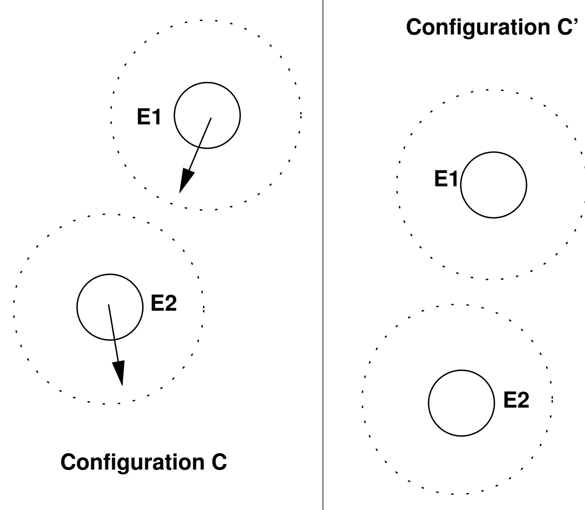

Fig. 1. An illustration of update independence. Two entities E_1 and E_2 are both about to move in different directions (as indicated by arrows) in a configuration C. Since their sensory and actuator ranges (indicated by dashed circles) within which they can affect their environment do not overlap, either agent can be removed in C and will end up in the same position in C' (on the right) if the reduced configuration is updated as when C is updated with both agents. Hence, E_1 and E_2 are mutually update-independent.

subconfigurations $\mathcal{C}$ *of* C *is* update-independent *if* $\forall C_1, C_2 \in \mathcal{C}\mathtt{UI}_{\mathtt{C}}(\mathtt{C}_1, \mathtt{C}_2)$. *A split* P^C_{Proc} *is update-independent if the set of all* Π_{C_i} *is update-independent.*

In other words, the presence of the other entity E_i cannot have any effect on E if its removal does not change the update of E. Note that update-independence is *not symmetric* (that's why we need the additional notion of "mutual update-independence"): it is possible that one entity E_1 is update-independent in C from another entity E_2, while the latter is not update-independent in C from the former (e.g., consider E_1 with maximum sensory and actuator ranges of 10 located in (0,0) and E_2 located in (0,50) with the same actuator range, but a circular sensory range of 100 for one sensor; then E_2 can sense E_1 and might change its behavior based on the perception, while E_1 is oblivious to E_2's presence). Moreover, update independence is not transitive either for obvious reasons, nor is it reflexive (e.g., an agent's

behaviors might or might not be completely independent of its own state).

Most importantly in the present context, update-independent configurations have the nice property that they can be directly "merged", i.e., the union of two update-independent configurations is itself a configuration with the following property:

Corollary 9 *Let C be a configuration, C' the successor configuration $C' = update(C)$, and Π_{C_1} and Π_{C_2} update-independent configurations obtained by splitting C via P^C_{Proc}. Then $C' = update(\Pi_{C_1}) \cup update(\Pi_{C_2})$.*

3.2 **P-ABM$_G$**– *A Generic Parallelization of ABMs*

The fact that update-independent configurations can be directly merged suggests a straightforward way to parallelize a given agent-based simulation with initial configuration C_0:

P-ABM$_G$ (C,$Proc$)

while notFinal(C) **do**

compute an update-independent split P^C_{Proc} for $Proc$

distribute each subconfiguration Π_{C_i} onto CPU_i in $Proc$

compute $(\Pi_{C_i})' := update(\Pi_{C_i}) \cup update(L)$ ($L \in first\ Loc_{\mathcal{M}}$) on each CPU_i

collect updates and merge into $C' := \bigcup(\Pi_{C_i})'$ (do not merge $L \in first\ Loc_{\mathcal{M}}$)

$C := C'$

$Proc := update(Proc)$

end while

The difference between entities and locations here is that entity updates are split, while location updates are replicated on each node. The assumption underlying this choice (e.g., as opposed to splitting locations as well) is that location updates are

typically very inexpensive compared to entity updates (e.g., the update of the "temperature" of a location might consist only of a simple increment of its previous temperature, while the update of an entity involves getting sensory readings, updating controller and body, and applying actuator effects). It is certainly possible to have simulations with complex updates of location properties, in which case it might make sense to extend splits to locations (i.e., treat locations the same way as entities). Similarly, updates of some entity types may be very cheap compared to other ones (or compared to the communication overhead their synchronization might cause), in which case it may be useful to *replicate* these entities in each parallel simulation, rather than treating them as part of the configuration that gets split (e.g., see the second demonstration experiment in Section 4.3).

It is a direct consequence of merging at the end of each update that the algorithm [12] is "step-wise correct" in the following sense:

Definition 10 (Step-wise Correctness) *Let $\mathcal{A}$ be a parallel algorithm for updating a spatial agent-based simulation $S_{C_0} = \langle C_0, C_1, C_2, \ldots, C_{final} \rangle$ of an S-ABM $\mathcal{M}$. $\mathcal{A}$ is* stepwise correct *if it produces a sequence of split configurations* $\langle \Pi_{C_0}, \Pi_{C_1}, \Pi_{C_2}, \ldots, \Pi_{C_{final}} \rangle$

[12] We would like to thank an anonymous reviewer for pointing out that it is not obvious that **P-ABM**$_G$ is, in fact, an algorithm without showing that it is always possible to compute a split. To see this, note that it is always possible to compute an update-independent split in the following inefficient, but trivial way: choose a split (at random), run the simulation in parallel for one step, and then compare the result to the simulation updated without a split (i.e., run on a single CPU): if the simulation states are the same, then the split was update-independent (repeat for all permutations). While this way of computing an update-independent split obviously defeats the purpose of parallelizing a model in the first place (as the whole simulation needs to be updated without being split), it shows that there is always a way of computing it, hence **P-ABM**$_G$ is an algorithm.

such that $C_k = \bigcup(\Pi_{C_k})$ *for all* $0 \le k < final$.

Corollary 11 **P-ABM**$_G$ *is step-wise correct.*

Note that the above algorithm is *adaptive* because the set of available CPUs is updated after every configuration update. Hence the algorithm can take the new set of resources (e.g., a larger number of available CPUs) into account when the new split is computed.

Aside from the question of how to compute an update-independent split, to which we will return shortly, it is clear that a parallelization of a simulation according to the above algorithm is only worthwhile if the cost of computing such a split, distributing subconfigurations and merging them subsequently is low compared to the cost of updating entities. At the same time, if updating an entity is very expensive, splitting entities based on update-independence might not be the best option in the first place. For example, if C consists of a large subconfiguration C_i of update-dependent entities, this configuration will be updated on one processor and thus incurs a computation cost linear in $|C_i|$, which is in the worst case $O(|C|)$. In such a case it is likely better to further split entities in $|C_i|$, distribute them over different CPUs, and use a mechanism to request and transfer the states of update-dependent entities in other subconfigurations as part of the update of an entity on the fly.

3.3 Towards Exploiting Properties of Spatial ABMs

As already mentioned, the important ingredient missing to be able to implement parallel algorithms like the above is an *efficient* way to detect update-independence. Detecting update-independence directly based on the definition of update-independence clearly defeats the purpose; for a given pool of processors *Proc* this would require

repeated computation of a split, independent update of all subconfigurations, and comparison of the merged updated subconfigurations to the update of the whole configuration in order to determine whether the split was update-independent. This means that the computational cost (in terms of space and time) of parallelizing and updating the subconfigurations in parallel is higher than computing the update of the whole configuration at once.

Fortunately, in spatial ABMs there is another criterion that is sufficient (but not necessary) for detecting the update-independence of two entities: being within each others' sensory and/or actuator range. For, clearly, entities that are not within sensory or actuator range of each other in a given configuration cannot possibly have any effect on each other, and are thus mutually update-independent.

Proposition 12 *Let E_1 and E_2 be two entities in a configuration C with locations Loc_1 and Loc_2, respectively, and let $Rng_1 = max\{Rng_S | S \in S_B \wedge S_B \in B_{E_1}\}$. If $Loc_2 \notin Rng_1$, then E_1 is update-independent of E_2 in C. The converse is not necessarily true.*

Note that being outside of each other's sensory range is a "conservative" estimate for mutual update independence, because two entities can still be update-independent even if they can sense each other (either because they do not take each perceptions of each other into account or because their perceptions coincidentally do not have any influence on the update in the particular context). In some cases, a finer-grained distinction may be possible and desirable (e.g., when a type of entity always ignores perceptions of its own kind). The general difficulty connected to any better derivation of potential interactions is, however, how to determine them automatically from the update functions, which may not be possible in a practical implementation if their representations are not explicitly accessible.

One crude way to support the automatic detection of potential interactions among entities across updates can be easily supported in metric environments, where the change of locations of entities is locally determined between successive configurations. For example, suppose the change in location in a metric spatial model is given in terms of speed, and the maximum speed of entities of type E is known, call it $maxspd_E$. Then the set of locations that E_j could influence or that could influence E_j at any cycle k after C_{i+k} $(k \geq 0)$ is given in terms of $maxspd_E \cdot k + max(Rng_E, Rng_{E^*})$ (the maximum sensory range of entities of type E and any other entity type E^*), which in a continuous 2D Euclidean environment will be the radius of an expanding circular region. Call this expanding subspace of the environment that results from the motion of an entity starting in a given configuration C_i the entity's "event horizon": [13]

Definition 13 (Event Horizon) *The* event horizon $\mathcal{EH}(E, C_i, k)$ *of an entity E starting in configuration C_i is the set of all locations after k updates within $maxspd_E \cdot k + max\{Rng_S | S \in S_{B_E}\}$ from its location in C_i, where $maxspd_E$ is the maximum change in location in one update cycle.*

---

[13] The term *event horizon* has been previously used in a slightly different sense in the domain of parallel simulation. E.g., "event horizon" in [17] refers to the set of events $\mathcal{E}$ that can occur before the first consequent event E' generated by an event $E \in \mathcal{E}$. Hence, it is the set of events $\mathcal{E}$ that can be safely executed in parallel, because no effects of any events in that set are seen during that time frame. This is similar to the way the term is used above, however, our usage refers to the first cycle an agent *could* affect another agent, rather than when it *will*.

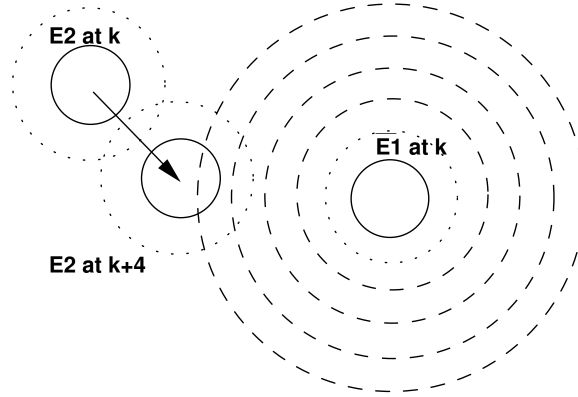

Fig. 2. An illustration of the event horizon. Entity E_2 moves from its position at cycle k to the new position at cycle $k+4$ (indicated by the arrow). The position of the entity represented by proxy entity $\overline{E_1}$ at cycle k is known, but not thereafter. The dashed circles indicate the increasing event horizon of that entity for subsequent cycles (including the maximum of the two sensory ranges–sensory ranges are indicated by dotted circles). At cycle $k+4$ E_2 intersects with the event horizon of $\overline{E_1}$ indicating that the actual position of E_1 is required before the update of E_2 can be computed.

Clearly, the event horizon of entities E in metric environments with $maxspd_E > 0$ is monotonically increasing, hence if $L \notin \mathcal{EH}(E, C_i, k+1)$, then $L \notin \mathcal{EH}(E, C_i, k)$. Moreover, it is symmetric, reflexive, but not transitive (which is important for computing dependencies among entities). Figure 2 shows the expanding event horizon in a 2D environment.

We can now refine the above algorithm for S-ABMs by merging only those subconfigurations that have update-dependencies across updates. The others can continue to update without merging. To determine which subconfigurations need to be merged and which can continue, we introduce the notion of a "proxy entity", which serves as a (local) placeholder in a subconfiguration for the last known state of an entity updated in another subconfiguration (on another processor). The main difference between proxy agents and non-proxy agents is their update function, which does not compute an update of the agent state (given that no information is avail-

able about the current state of the agent represented by the proxy agent), but rather returns the event horizon of the proxy entity based on a set of locations:

Definition 14 (Proxy entity) *A* proxy entity $\overline{E}$ *of an entity E (in the following always denoted by a bar) consists of the entity's body and controller, but with a different update function* $\overline{E} = \langle B_E, C_E, \overline{U_E} \rangle$*, where* $\overline{U_E} : \mathcal{P}(Loc_{B_E}) \mapsto \mathcal{P}(first\ Loc_{\mathcal{M}})$*.* [14]

Proxy entities have mere *representational functions* and cannot be updated like regular non-proxy entities (i.e., they cannot change their state across configurations). Yet, they can be used to compute the event horizon of the entity in subsequent configurations based on the last known configuration at which the proxy entity was updated by repeatedly applying $\overline{U_E}^k(\{Loc_{\overline{B_E}}\})$ to the last known location Loc_{B_E}. That way, given the state of a proxy-entity $\overline{E_j}$ (representing an entity E_j in subconfiguration C_j) it is is possible to determine the maximum possible subspace of the environment on which an entity E_j in configuration C_i could exert any influence in subsequent updates of C_i and thus the number of updates of C_i (based on the known states and state changes of entities in C_i) before any interaction between E_j and any $E_i \in C_i$ is possible.

We can now state an important lemma:

Lemma 15 (Interaction Lemma) *Let* C_1 *and* C_2 *be two subconfigurations of a configuration C containing only non-proxy entities and let* C_1^* *and* C_2^* *be the configurations obtained from* C_1 *and* C_2 *by adding the proxy entities in* $\overline{Ent_2} \in C_2$ *and* $\overline{Ent_1} \in C_1$ *that represent the states of some non-proxy entities* $Ent_2 \in C_2$ *and* $Ent_1 \in C_1$*, respectively. Moreover, let n be the largest number such that no non-*

[14] We will extend the bar notion of proxy entities to sets of proxy entities (e.g., if *Ent* is a set of entities, then $\overline{Ent}$ is a set of proxy entities obtained from the entities in *Ent*).

proxy entity $E_1 \in C_1$ *has* $Loc_{E_1} \in \mathcal{EH}(\overline{E_2}, C_1, n)$ *for any* $\overline{E_2} \in \overline{Ent_2}$ *and no non-proxy entity* $E_2 \in C_2$ *has* $Loc_{E_2} \in \mathcal{EH}(\overline{E_1}, C_2, n)$ *for any* $\overline{E_1} \in \overline{Ent_1}$*. Then for all* $k \leq n$, $U^n_{\mathcal{M}}(C_1) \cup U^n_{\mathcal{M}}(C_2) = U^n_{\mathcal{M}}(C_1 \cup C_2)$*. Or put differently,* C_1 *and* C_2 *are mutually update-independent for at least the first n updates.*

PROOF. Suppose there is a $k \leq n$ such that $U^k_{\mathcal{M}}(C_1) \cup U^k_{\mathcal{M}}(C_2) \neq U^k_{\mathcal{M}}(C_1 \cup C_2)$, i.e., C_1 and C_2 are not update-independent after k updates. Then by Def. 8, there must be two entities E_1 and E_2 that are update-dependent. It follows by Prop. 12 that one of the two entities must be within sensory range of the other. Without loss of generality, assume that E_1 is within sensory range of E_2 and that $E_1 \in C_1$ (the case for E_2 within sensory range of C_1 and $E_2 \in C_2$ is analogous). Then $E_2 \in C_2$ (as otherwise $update^k(C_1) \cup update^k(C_2) = update^k(C_1 \cup C_2)$). Now consider the proxy representation $\overline{E_2}$ of E_2 in C_1 (based on Def. 14). Since E_1 is within sensory range of E_2 after k updates of C, E_1 must be within the event horizon of $\overline{E_2}$ after k updates by Def. 13. But, by assumption, there is no non-proxy entity $E_1 \in C_1$ such that $Loc_{E_1} \in \mathcal{EH}(\overline{E_2}, C_1, n)$ for any $\overline{E_2} \in \overline{Ent_2}$, hence it cannot be in $\mathcal{EH}(\overline{E_2}, C_1, k)$ either. Contradiction.

3.5 **P-ABM**$_S$*– A Parallelization for Spatial ABMs*

The *Interaction Lemma* confirms that two mutually update-independent subconfigurations C_1 and C_2 can be updated independently as long as none of the event horizons of the proxy entities in either configuration contains a location of a non-proxy entity in that configuration. When such a configuration is reached, the actual state of the entity represented by the proxy entity needs to be obtained. Hence, we can formulate the following refined version of **P-ABM**$_G$ for S-ABMs:

P-ABM$_S$ (C_0,$Proc$)

$oldProc := \emptyset$

$k := 0$

while notFinal(C_k) **do**

if $oldProc \neq Proc$ **then**

compute an update-independent split $P^{C_k}_{Proc}$ for $Proc$

distribute each configuration $\Pi_{C_{k,i}}$ onto CPU_i in $Proc$

$\Pi^*_{C_{k,i}} := \{\overline{\Pi_{C_{k,j}}} | \Pi_{C_{k,j}} \in P^{C_k}_{Proc} \wedge i \neq j\} \cup \{\Pi_{C_{k,i}}\}$

$oldProc := Proc$

end if

compute all $\mathcal{EH}(\overline{\Pi_{C_j}}, C, k)$ for the last known state from some configuration C

for proxy entity E_j that has a non-proxy entity within $\mathcal{EH}(\overline{\Pi_{C_j}}, C_k, k)$ **do**

get state of E_j at k from CPU_j and update $\overline{E_j}$

end for

compute $(\Pi^*_{C_{k,i}})$' := $update(\Pi^*_{C_{k,i}}) \cup update(L)$ ($L \in first\ Loc_{\mathcal{M}}$) on each CPU_i

update $Proc$

if $oldProc \neq Proc$ **then**

merge all $C_{k+1} := \bigcup U_{\mathcal{M}}(\Pi_{C_{k,i}})$

end if

$k := k+1$

end while

The main difference between **P-ABM**$_S$ and **P-ABM**$_G$ is the potential for CPUs to update asynchronously as long as the set of entities running on them is update-independent of the rest. And even when there are potential interactions, as determined by the proxy entities' event horizons, simulations do not have to be merged, but only proxy entities need to be updated based on the communicated locations of the non-local entities they represent. Consequently, it is not necessary to compute

new update-independent splits either before every update cycle. Simulations are, however, still merged and splits are recomputed (as with **P-ABM**$_G$) if the processor pool *Proc* changes, thus preserving the adaptiveness in **P-ABM**$_S$.

We can now state the step-wise correctness of **P-ABM**$_S$ (for space reasons, we only provide a proof sketch):

Theorem 16 **P-ABM**$_S$ *is step-wise correct.*

PROOF. [Sketch] By induction on the length n of the simulation. For $n = 0$, nothing needs to be shown, the algorithm will return right away as the initial configuration $C_0 = C_{final}$ is also the final configuration.

Now suppose the algorithm is step-wise correct up for n updates. In particular, suppose $C_n = \bigcup(\Pi_{C_n})$. We need to show that $C_{n+1} = \bigcup(\Pi_{C_{n+1}})$. First observe that this it trivially true if $C_n = C_{final}$, so assume $C_n \neq C_{final}$ and suppose $C_{n+1} \neq \bigcup(\Pi_{C_{n+1}})$. Then there must be at least one non-proxy entity E_π in one subconfiguration in $\Pi_{C_{n+1}}$ whose state differs from the state of the corresponding entity E in C_{n+1}. Since their states were the same at cycle n by induction hypothesis, their perceptions at cycle $n+1$ must have been different, giving rise to different states after their update. The only entities with respect to which the perception of E_π and E can differ are proxy entities. Since **P-ABM**$_S$ computes the event horizons of all proxy entities in all subconfigurations to check whether they contain locations of non-proxy entities according to Lem. 15, it is not possible for E to have a new interaction with some entity E_j that was not detected by E_π and would have eventually [15] led to an update of the location of the proxy entity $\overline{E_j}$ reflecting the latest

[15] Note that if two update-independent subconfigurations are updated asynchronously in parallel on two CPUs, then it may be possible that CPU$_\pi$ running E_π either lags behind

state of E_j ("for-loop"). Contradiction. Hence, $C_{n+1} = \bigcup(\Pi_{C_{n+1}})$.

4 Implementation of P-ABM$_S$ and Experimental Results

We implemented **P-ABM$_S$** in our agent-based SWAGES environment to be able to test its effectiveness for automatically parallelizing (spatial) agent-based simulations. To demonstrate the performance of **P-ABM$_S$**, we employed two tasks, a simple *swarm task* (agents are based on previous work [16,2]), in which all swarm agents must locate checkpoints in their environment and gather at them as quickly as possible, and a more complex environmental traversal task, where agents must traverse an obstacle environment with moving obstacles as quickly as possible without collisions (agents are again based on previous work [18,19]). The two tasks are intended to illustrate two different kinds of scenarios, where (automatic) parallelizations of S-ABM runs can pay off. We first give a very brief overview of the implementation of **P-ABM$_S$** in SWAGES, followed by short summaries of the employed agent models, their tasks, and performance results from simulation runs with **P-ABM$_S$** compared to the sequential version. The discussion concluding this section then points to the time savings one can expect from employing **P-ABM$_S$** based on the employed agents in S-ABMs.

CPU$_j$ running E_j in terms of completed update cycles k or is ahead. In the former case, CPU$_\pi$ will have to wait in order to update $\overline{E_j}$, in the latter it can update $\overline{E_j}$ right away as no interaction of any of its entities could have occurred before k, see Section 4.1 for an example.

4.1 *Implementation of* **P-ABM**$_S$ *in* SWAGES

SWAGES is an agent-based simulation and experimentation environment intended for any kind of computing environment (e.g., from homogeneous Beowulf clusters to heterogeneous computers connected only via the Internet).[16] It consists of several distributed components that cooperate closely to achieve maximum resource utilization in a heterogeneous dynamically changing computing environment. For the present context, two components are most relevant: the simulation environment SIMWORLD, and the GRID SERVER scheduling and monitoring the execution of simulations. SIMWORLD is an agent-based simulation environment implemented on top of the SIMAGENT toolkit [20] and has been used for many different agent-based models with a variety of agent ranging from very simple reactive (e.g., [10]) to fairly complex deliberative agents (e.g., [19]). The GRID SERVER provides the communication infrastructure to start, run, and supervise simulations in SIMWORLD, gather the results and store them in an easily accessible way for statistical analysis. The server can schedule sets of simulation experiments (e.g., simulations with a variety of different initial conditions) and ensure their timely completion by monitoring their performance and detecting problems with the execution (e.g., because the load on a host is too high, or the simulation crashed), in which case it can take any number of recovery actions (from resuming a simulation on a different host if its state was saved, to restarting it anew if no state information was available). Each SIMWORLD instance can run on its own host and has its own representation

[16] SWAGES can run on any computer for which the poplog environment `http://www.poplog.org/` has been compiled, i.e., typically Linux and Solaris machines. It can be used on any host on which the experimenter has an account without any pre-installed software or daemons other than the standard ssh daemon, see `http://www.nd.edu/~airolab/software/`.

in the server in the form of a SIMCLIENT that entertains a socket connection to its SIMWORLD instance for all communication purposes (e.g., information about the current simulation cycle, simulation parameters, etc. will be delivered on this connection).

To implement **P-ABM**$_S$ in SWAGES, both SIMCLIENT and SIMWORLD had to be modified to accommodate *proxy entities* and to update their states. Specifically, the server representations of each SIMWORLD instance needed to be augmented by tables to store the state information of their proxy entities as well as synchronization primitives to allow them to access and update the tables in a coordinated fashion. Effectively, each SIMWORLD instance computes at the beginning of each update cycle the event horizons of all proxy entities to determine if there are potential interactions, at which point a *request* is sent for each required proxy entity's update to the GRID SERVER*via* the SIMCLIENT socket (the "FOR"-loop in **P-ABM**$_S$). With the request for update, the current state information of non-proxy agents is sent as well, which gets stored in a table shared by all SIMCLIENT instances of a distributed simulation. Each SIMCLIENT then blocks on its specific requests and is awakened whenever updates are delivered by clients running the respective non-proxy entities. Each update consists of an ID of the non-proxy agent (which is unique across all SIMWORLD instances), the status information and the cycle at which it was obtained. If the cycle of the updated information is before the requested one, a SIMCLIENT will block again waiting for later information. Otherwise it will use the information with the future cycle number *closest to its current cycle*.

At first glance, this may seem incorrect–why should it be sufficient for *entity15* in SIMWORLD instance 4 requiring the update of its proxy representation *proxy-entity64* (of *entity64* in SIMWORLD instance 7) at *cycle 321* to use its state at *cycle*

598, for example, which is the closest available future state in the shared table? The reason is that all SimWorld instances perform checks for both proxy and non-proxy agents. Hence, instance 7 will perform the “reverse check” of the check performed by instance 4, namely whether *entity64* is within the event horizon of *proxy-entity15* and determine that this is the case only at cycle 598. By the *Interaction Lemma*, it follows that for any cycle between 321 and 598 *entity64* was not in the event horizon of *proxy-entity15*, hence, by symmetry, it *could not have been* within sensory range of *entity15* until then either (because the event horizon of an entity is the maximum possible sphere of influence of that entity). The fact that instance 4 requires that information earlier simply means that *entity15* was within the event horizon of *proxy-entity64* at cycle 321 based on the last information in instance 4 about the location of *entity64*, but obviously *entity64* in instance 7 must have moved in a different direction that did not require an update of *proxy-entity15* until cycle 598. Hence, instance 4 will update the location of *proxy-entity64* with the location of *entity64* at cycle 598 and consequently ignore *proxy-entity64* for all update cycles up to cycle 598 (as no interactions with it could have occurred before).

4.2 The Swarm Gathering Task

In this task, agents must locate checkpoints in their environment and gather at the nearest checkpoint as quickly as possible.

4.2.1 Agent Model

All agents have sensors to detect checkpoints within a 360° radius, whose range was set to 400 units in the experiments described below. The agent update function

(in the entity type definition) is described as follows (see [16] for a more formal description):

Rule 1: if no checkpoint is sensed, perform a random walk (described below)

Rule 2: if some checkpoints are sensed, go directly towards the closest checkpoint

Rule 3: if some checkpoint is within gathering range, do not move

When agents are in random walk mode, they move in one direction for 200 cycles (unless a checkpoint comes within sensory range), then turn randomly between -45 and 45 degrees and continue for another 200 cycles, and so on. The update algorithm for reactive agents is as follows:

Reactive($agent$, $checkpointList$)

$closest$:=infinitely-far-checkpoint

for all $C \in checkpointList$ **do**

  if $distance(agent, C) < distance(agent, closest)$ **then**

    $closest := C$

  end if

end for

if $closest$ ==infinitely-far-checkpoint **then**

  $randomWalk(agent)$

else

  $moveToward(agent, closest)$

end if

4.2.2 Simulated Environment

The environment used in the experiments consists of a continuous 2D plane, in which 16 clusters are placed regularly on a grid within a 32000 x 32000 square unit area (for comparison, agents have a circular body of diameter 5). Each cluster

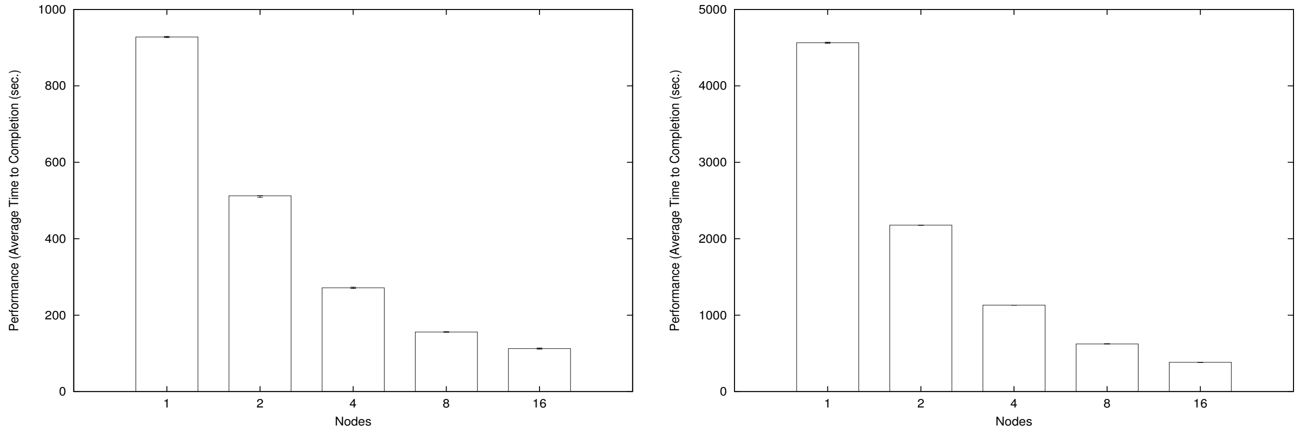

Fig. 3. Average time to completion for 1, 2, 4, 8, and 16 nodes for the *swarm task*, where error bars denote confidence intervals for $\alpha = 0.05$ (left) and the *cluster setup* of the obstacle avoidance task (right).

contains 128 agents randomly placed in within a radius of 400 around the center, and 16 checkpoints randomly placed within a radius of 200 around the center. To test the gain in parallelizing simulations in a scenario that plays to the strengths of the algorithm, the clusters were separated by enough distance so that the event horizons of agents across clusters do not intersect in the course of a given simulation of 100 cycles. Moreover, the setup is such that update-independent splits can be computed (in this case the split has to be computed only once at the beginning) and large subsets of update-independent agents can be run on separate CPUs.

4.2.3 Simulation Results

The results reported are averages over 20 simulation runs of 100 cycles each. Each of the 20 initial conditions was simulated using 1, 2, 4, 8, and 16 nodes in a dedicated Linux cluster of dual 2.4GHz Xeons with 1GB RAM (the simulations require less than 100MB of memory). The times reported include all overhead of starting and finishing SWAGES, as well as distributing the simulations when more than

one node is used.

Figure 3 presents the performance results. Initially, increasing parallelism is very effective; doubling the nodes available cuts the total time nearly in half. As the number of nodes increases, however, we find decreasing benefit. The difference between 8 nodes and 16 nodes is not nearly half, indicating that the overhead of parallelization is nearing the computation being performed on each CPU. In fact, the initial split for the 16-node configuration is not complete until roughly 30 seconds have elapsed, a sizable portion of the 112.5 average. Still, the results show that the time savings of the parallel algorithm in well-suited scenarios scales almost linearly with the number of employed CPUs, a scaling factor one would hope for.

4.3 The Dynamic Obstacle Avoidance Task

In this task, agents must locate the checkpoint in their environment and move to it as quickly as possible. Unlike the previous task, however, there is a hazard added to the environment: moving obstacles. Contact with these obstacles is fatal for agents, so they must traverse the environment while avoiding the obstacles. With stationary obstacles, simple reactive agents (such as the ones in the previous task) perform fairly well, although not as well as deliberative agents that can plan a route [19]. Moving obstacles make the environment very difficult for the reactive agents, making deliberative agents a better choice for the dynamic obstacle avoidance task.

4.3.1 Agent Model

The deliberative agents used here are extensions of the reactive agents. Their sensors have a range of 1600 units, which covers the whole environment to force the algorithm to update the state of proxy agents before each cycle. Moreover, agents have a simple route planning mechanism which allows them to find a route to the nearest checkpoint, avoiding obstacles. The planner is based on a simplified version of the A^*_ε algorithm [14]. A^*_ε is a variant of A^* in which the cost of the solution returned is guaranteed to be no greater than $1+\varepsilon$ times the cost of the optimal solution. A^*_ε is a good compromise for a route planning agent between quality of the route and computation time in dynamic environments, as good rather than optimal plans are often sufficient. Deliberative agents also maintain memory components in which they store information about the locations of checkpoints and obstacles. The relative positions of entities in memory are updated as the agent moves, and a "coherence mechanism" checks whether entities within sensory range are in the same locations as recalled stored entities. Once an agent has made a plan to reach a checkpoint while avoiding obstacles, it follows the plan to the checkpoint unless there is a change in the locations of the obstacles. For the dynamic obstacle avoidance task, this means that agents will require frequent re-planning (depending on the speed of the moving obstacles).

The agent update function for deliberative agents is similar to reactive agents:

Rule 1: if some checkpoint is within gathering distance, do nothing

Rule 2: if no checkpoint is sensed, perform a random walk

Rule 3: if there is no plan, plan a route to the nearest checkpoint

Rule 4: if sensed obstacles do not match memory or a nearer checkpoint is detected, delete the current plan

Rule 5: if there is a valid plan, execute the next step in the plan

The update algorithm for deliberative agents is as follows:

Deliberative($agent$, $checkpointList$, $obstList$)

if $distance(agent, agent.planTarget) > gatheringDist$ **then**

  $agent.memory := updateMemory(oldLoc, agent, agent.memory)$

  $rePlan := false$

  for all $E \in obstList$ **do**

    $found := false$

    for all $M \in agent.memory$ **do**

      if $distance(E, M) < \varepsilon$ **then**

        $found := true$

      end if

    end for

    if $found == false$ **then**

      $insert(E, agent.memory)$

      $rePlan := true$

    end if

  end for

  for all $C \in checkpointList$ **do**

    if $distance(agent, C) < distance(agent, agent.planTarget)$ **then**

      $rePlan := true$

      $agent.planTarget := C$

    end if

  end for

  if $agent.plan == NULL$ **then**

    if $agent.planTarget == NULL$ **then**

      $randomWalk(agent)$

    else

$agent.plan := makePlan(agent, agent.planTarget)$

end if

else

if $rePlan == true$ **then**

$agent.plan == NULL$

else

$executePlanStep(agent, agent.plan)$

end if

end if

end if

4.3.2 Simulated Environment

The environment used in these experiments consists of a continuous 2D plane of size 1600 x 1600 units. There are 40 moving obstacles randomly placed throughout the environment. These obstacles are initialized with random headings, and move in a straight line unless they are at the boundary of the environment, in which case, they “bounce off.” There are two experimental setups: clustered and random. In the clustered setup, there is a single checkpoint located at (750,0), and a group of 16 agents tightly clustered at (-750, 0). These agents have very similar views of the environment, so, given the control algorithm above, they are likely to plan similar routes at the same time. Thus, when any agent plans during a cycle, typically all other agents plan during that cycle as well. In the random setup, agents are instead placed randomly in the left half of the environment (see Figure 4). These agents have rather different views of the environment, so they will frequently plan at different cycles. In both clustered and random setups, the agents’ event horizons extend to the entire environment (given the sensory range of the agents). Hence,

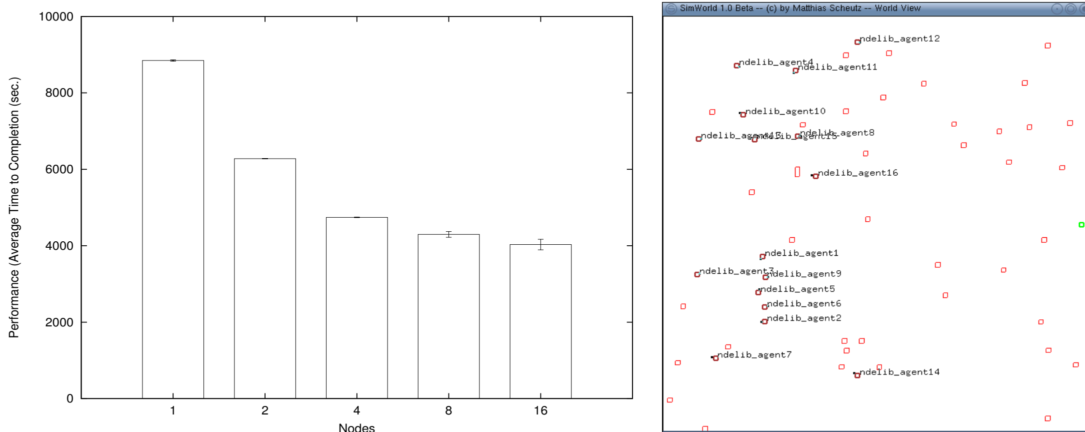

Fig. 4. Average time to completion for 1, 2, 4, 8, and 16 nodes (left)– error bars denote confidence intervals for $\alpha = 0.05$). A screen shot of the random setup of the obstacle avoidance task, where circles with names indicate deliberative agents, the green circle without a name indicates the target location, and squares indicate moving obstacles (right).

each agent requires updates of all other agents at each cycle. Note that obstacles are treated as "replicated entities" in both setups due to the low computational cost of their update.

4.3.3 Simulation Results

The details of the obstacle avoidance experiments are the same as for the swarm task: performance results represent the average over 20 simulation runs of 100 cycles each. Results are reported for 1, 2, 4, 8, and 16 nodes in the dedicated Linux cluster described above, and include the overhead of starting and finishing SWAGES and distributing the simulations.

The performance results for clustered setups on the dynamic obstacle avoidance task are presented in Figure 3. Here we find that increasing parallelism is consistently effective all the way up to 16 nodes (i.e., one agent per node). In each

case, doubling the number of CPUs cuts the average task completion time roughly in half. In contrast to the swarm task, the computation for the deliberative agents greatly outstrips the communication overhead, even though the agents are moving in lockstep, requiring updates from every agent at every cycle.

Random setups do not perform as well, however. Figure 4 shows that parallelization is not very effective and that the situation gets worse as the number of CPUs increases. Like the clustered setup, these agents require updates from all other agents every cycle, but unlike the cluster setup they do not all plan at the same time. Hence, it frequently happens that several agents are only executing plan steps, while a few others (in the worst case just one) need to plan. Because the updated positions of all agents are needed at the beginning of all cycles, agents executing plan steps will still have to wait for the planning agents to finish their plan. In such setups, the sum of the durations of update cycles in which at least one agent needs to plan is a theoretical lower bound on the shortest overall runtime. Hence, the small gains obtained by parallelizing such simulations (compared to single CPU runs) might not be worth the cost of tying up too many CPUs. We will make this point more precise in the next section.

4.4 Performance of $\mathbf{P\text{-}ABM}_S$

Overall, $\mathbf{P\text{-}ABM}_S$ demonstrates good performance in both types of tasks–asynchronous updates with many simple agents and synchronous update with complex agents–when the computation times required to compute the update functions of all entities is about the same. With strongly heterogeneous update times (as in the random setup), savings are only marginal. Hence, the question arises what savings in computation time we can in general expect in $\mathbf{P\text{-}ABM}_S$ (or any parallel algorithm for

spatial agent-based models, for that matter).

The cost of updating a configuration C_k given *Proc* in **P-ABM**$_S$ (assuming the split has been computed and the entities are already distributed accordingly to the CPUs in *Proc*) is given by:

$$max\{\tau_k(\Pi_{C_{k,i}})|\Pi_{C_{k,i}} \in P^{C_k}_{Proc}\} + \{\tau_k(o)|replicate(o) \vee proxy(o) \vee location(o)\}$$

where the first factor is the update time of subset of entities on one CPU that takes longest, the second is the cost of updating replicated entities, proxy entities, and locations ($\tau(x)$ denotes the update time of an entity x or set of entities x). We can compare this to the cost of updating on a single CPU, which is given by:

$$\sum_e \{\tau_k(e)|e \in C_k\} + \{\tau_k(o)|replicate(o) \vee proxy(o) \vee location(o)\}$$

It is clear that the longest entity update determines the overall performance of **P-ABM**$_S$. E.g., in the above random setup of deliberative agents, one agent that needs to compute a complex plan, while all the other ones just execute their respective plans, exclusively determines the overall cost of updating a configuration, given that the sum of the respective plan execution times is only a small fraction of the computation of a new plan. Hence, there are effectively only marginal savings in parallelizing simulations in principle when a single agent dominates the computational cost of updating a given configuration. [17]

[17] If the dominating agent does not stay the same through a sequence of updates, then it may be possible to obtain some savings by speculative updates of configurations that do not contain the dominating agent and backtracking later if it turns out that these configurations

On the other hand, if the update times of single agents are more or less balanced from configuration to configuration (e.g., as with the reactive agents in the swarm task or the cluster setup for the deliberative agents), then the longest parallel update will be approximately $1/|Proc|$ (plus the overhead for synchronizing proxy agents and possibly for re-computing splits) compared to the single update. Hence, the savings will be approximately linear in $|Proc|$ in the best case (e.g., with little communication overhead and no additional splits). Moreover, **P-ABM**$_S$ (and even **P-ABM**$_G$) will be able to reduce the overall runtime of simulations whenever the cost of updating entities is larger than the cost of communication and synchronization across nodes (e.g., as in the case of the traversal task, where agents synchronized after each update due to intersecting even horizons).

In sum, the nature of the agents and the computational costs of their update functions in different environmental situations will essentially determine the performance of the parallelization, from the worst-case scenario with single agents dominating the cost of a whole ABM run, to the best case scenarios of agents whose updates require about the same computational cost from configuration to configuration (as demonstrated above).

5 Related Work

While many different ABM simulation environments have been proposed, most of which are targeted at specific types of agent-based models (e.g., economic, Alife, swarms, etc.), only a few of them support the distribution of simulations over multiple hosts. To our knowledge, all ABM simulation environments capable of dis-

were not update-independent of the configuration containing the dominating agent. The exact details and savings of this approach will have to be left for future investigations.

tributed simulation implement some form of stepwise synchronization mechanism similar to **P-ABM**$_G$, but without support for the automatic and dynamic distribution of agent-based models. None of these simulation environments utilize the spatial information present in S-ABMs.

Som and Sargent present a load balancing algorithm based on *strong groups* [21]. A strong group is a collection of logical processes that frequently interact with each other, but infrequently interact with logical processes in other strong groups. This is similar to the notion of distributing agents based on update-independence, as described above. Som and Sargent point out that load balancing algorithms based on utilization can be problematic in some cases. Strong groups, by definition, do not interact often, so it may take a while to detect when an interaction requires a rollback, leading to wasted time. At times, it is better to balance the load such that a fast-executing strong group is *slowed down*, so that it does not get too far ahead; the resources can then be used productively by some other strong group. [18]

Wang et al. introduce the *Interaction Resolver* to enforce mutual exclusion in distributed simulations [22]. The Interaction Resolver is a middle-ware component that ensures consistency throughout the simulation by protecting access to shared state. When an interaction occurs between agents, the agent with the highest priority is allowed to make its change to the state, and the losers are required to roll back to previous states. Interaction detection is performed each cycle, thus requiring agents to execute in lockstep.

The *Syncer* [23] system was used to implement a distributed version of *Swarm* [24] (one of the major toolkits for SWARM-based ABMs). A syncer component con-

[18] We are investigating ways to incorporate this idea into our algorithms to improve dynamic split computations.

trols the advancement of time in a simulator or another syncer. Thus, the system allows hierarchical systems of distributed simulations. The syncer sends its subordinates messages indicating that they may begin execution, and at what point they must stop. The simulation informs the syncer when it is done with those cycles, and can send *time synchronized information* (TSI) messages to other simulations. A new class `SyncerSwarmImpl` that Swarm models must extend instead of `SwarmImpl` acts as an interface to the rest of the Syncer hierarchy, receiving the `start` and sending the `done` messages, as well as forwarding TSI messages. Remote objects are represented by proxies, sharing the same interface and forwarding requests to the remote object via TSIs. Syncer includes facilities that allow simulations to update asynchronously, however, it is unclear whether they are used in the Swarm implementation. In any case, dynamic distribution is not possible with Syncer/Swarm.

MACE3J is a Java-based distributed simulation environment [25]. It is not committed to any particular simulation environment, but strives to be a general-use testbed for multi-agent simulations. MACE3J provides services, such as registration, scheduling, and messaging to *ActivationGroups*, which consist of *ActiveObjects*, the representation of agents in the system. The goal of the project is to provide a flexible testbed for simulation research. Multiple event-control mechanisms are supported, data-gathering hooks are built into the system, and reusable ActiveObjects components, and importers for components from other systems are provided.

Other work on parallel agent-based simulations is based on HLA, the *High Level Architecture* framework for simulator reuse and interoperability developed by the US DoD's Defense Modeling and Simulation Office. HLA groups simulations together into a *federation*, which consists of the simulations (the *federates*), the Federation Object Model (FOM), which defines the interfaces between simulations,

and the Runtime Infrastructure (RTI), through which federates communicate.

HLA_REPAST [26] uses HLA to distribute simulations based on the *RePast* toolkit [27]. RePast models must implement a discrete event scheduling engine that defines the execution of events in the model. The RePast "executive" executes events in this scheduler to run the simulation. HLA_REPAST extends the executive to include two schedulers, one for the local objects, and one for the remote objects. Each of these can affect the local model's state. The external scheduler receives input from the RTI, and the local scheduler sends output to the RTI. HLA_REPAST simulations also proceed in lockstep, requiring the local scheduler to wait for all events with earlier timestamps to arrive from the RTI before executing a local event, and there is no facility for dynamic distribution of simulations.

The model closest to our SWAGES environment in terms of the implementation platform is HLA_AGENT [28], an implementation of the SIM_AGENT toolkit [20] that allows SIM_AGENT simulations to be distributed. SIM_AGENT is extended to include information about the FOM, the agents to be executed in the current federate, and proxies for agents executing in remote federates. The scheduler is modified such that only local agents are run. Object creation and deletion are achieved via wrappers that register new objects with the RTI and allow calls for agent destruction, also via the RTI. Federates can "subscribe" to the external attributes of an agent, so that they are notified when they change. The simulations proceed in lockstep, synchronizing at the beginning of each cycle. This is a major difference with the work described here, where the semantics of the event horizon are relied upon to ensure that updates occur only as needed. Also, HLA_AGENT does not support dynamic distribution of simulations.

Logan and Theodorpoulos introduce the notion of a *sphere of influence* [29]. An

event's sphere of influence is the set of shared variables immediately affected by the event. The calculated spheres of influence can be used to derive a decomposition of the simulation into *logical processes* that can be executed in parallel. When the spheres of logical processes overlap, information is shared via separate *communication logical processes*, which host shared state variables in a hierarchical tree structure. Communication processes can split as the load increases, and simulation logical processes can "migrate" to another communication process if more of its sphere of influence is hosted there. While these spheres of influence are similar to the event horizons described above, there is an important difference: whereas the sphere of influence represents the *actual* effect of an action (from the privileged perspective of the process in which the event took place), the event horizon represents the *possibility* of an effect from the perspective of another process. This allows distributed processes to postpone updating shared state until there is the potential for interaction.

6 Conclusion

We have proposed a formal framework for describing agent-based models, in particular, spatial agent-based models and presented two algorithms for the automatic and adaptive distribution of agent-based simulations specified in the framework over multiple CPUs. The first algorithm $\mathbf{P\text{-}ABM}_G$ works for any agent-based simulation specifiable in our formalism, the second algorithm $\mathbf{P\text{-}ABM}_S$ utilizes the additional information available about the sphere of influence of agents or entities in spatial agent-based models such as SWARMS, ANTS, and many others. We showed that it is possible to exploit this information to allow for asynchronous updates of distributed simulation instances, which only have to synchronize whenever

a non-local entity could potentially influence a local entity and vice versa as measured in terms of the maximum sphere of influence of an entity over time (i.e., its event horizon). We showed that the algorithm is correct in that it yields the exact same results as the non-parallel version for all simulations. We also reported results from a preliminary implementation of the algorithm in our SWAGES environment that demonstrate the utility of the algorithm for swarm-like simulations as well as simulations with complex agents, where the update function for each agent takes up a significant amount of computation time. It is worth noting that we achieved the significant gains in execution time in our empirical evaluations despite the very inefficient preliminary implementation of **P-ABM**$_S$, which is currently only a "proof-of-concept" implementation and thus not optimized at all (e.g., all updates are stored in the GRID SERVER, which creates a communication bottleneck; ideally, all updates and request would take place *via* peer-to-peer connections of the SIMWORLD instances). We expect an optimized implementation of the process synchronization mechanism to reduce the communication overhead substantially.

In addition to improving the inefficiencies of the current implementation, future work will investigate the performance of the proposed algorithms with dynamically changing pools of available CPUs (which will include a study of the effect of merging subconfigurations and recomputing update-independent splits). Moreover, we plan to explore different methods for obtaining better estimates of the event horizons based on reflective inspections of the agents' sensory update functions (i.e., by deriving from the agent model the exact context in which agents can interact). We expect this to lead to significant improvements in cases where interactions within sensory range only occur in a limited subset of agent states. Finally, it will be worth investigating speculative execution in the context of **P-ABM**$_S$, which could lead to

improved performance given the conservative estimates of possible intersections of event horizons as long as the cost of saving the subconfigurations (and rolling back to them when a speculative execution failed) is not too expensive.

References

[1] C. W. Reynolds, Flocks, herds, and schools: A distributed behavioral model, Computer Graphics 21 (4) (1987) 25–34.

[2] M. Scheutz, P. Schermerhorn, Many is more but not too many: Dimensions of cooperation of agents with and without predictive capabilities, in: Proceedings of IEEE/WIC IAT-2003, IEEE Computer Society Press, 2003.

[3] V. Trianni, T. H. Labella, M. Dorigo, Evolution of direct communication for a swarm-bot performing hole avoidance, in: Proceedings of the 4th Intl. Workshop on Ant Colony Optimization and Swarm Intelligence, 2004, pp. 131–142.

[4] P. Schermerhorn, M. Scheutz, The effect of environmental structure on the utility of communication in hive-based swarms, in: IEEE Swarm Intelligence Symposium 2005, 2005.

[5] P. Schermerhorn, M. Scheutz, The utility of heterogeneous swarms of simple uavs with limited sensory capacity in detection and tracking tasks, in: IEEE Swarm Intelligence Symposium 2005, 2005.

[6] S. S. Andrews, D. Bray, Stochastic simulation of chemical reactions with spatial resolution and single molecule detail, Physical Biology 1 (137-151).

[7] T. S. Shimizu, The spatial organisation of cell signalling pathways - a computer-based study, Ph.D. thesis, University of Cambridge (2002).

[8] V. Grimm, Ten years of individual-based modelling in ecology: what have we learned

and what could we learn in the future?, Ecological Modelling 115 (2-3) (1999) 129–148.

[9] S. F. Railsback, B. C. Harvey, R. H. Lamberson, D. E. Lee, N. J. Claasen, S. Yoshihara, Population-level analysis and validation of an individual-based cutthroat trout model, Natuaral Resource Modeling 15 (1) (2002) 83–110.

[10] M. Scheutz, P. Schermerhorn, Predicting population dynamics and evolutionary trajectories based on performance evaluations in alife simulations, in: Proceedings of GECCO 2005, 2005.

[11] B. J. L. Berry, L. D. Kiel, E. Elliot, Adaptive agents, intelligence, and emergent human organization: Capturing complexity through agent-based modeling, Proceedings of the National Academy of Science 99 (2002) 7187–7188.

[12] R. Conte, Agent-based modeling for understanding social intelligence, Proceedings of the National Academy of Science 99 (2002) 7189–7190.

[13] P. Schermerhorn, M. Scheutz, Implicit cooperation in conflict resolution for simple agents, in: Agent 2003, 2003.

[14] J. Pearl, A^*_ε—an algorithm using search effort estimates, in: IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 4, 1982, pp. 392–399.

[15] M. Scheutz, G. Madey, S. Boyd, tMANS–the multi-scale agent-based networked simulation for the study of multi-scale, multi-level biological and social phenomena, in: Proceedings of Spring Simulation Multiconference (SMC 05), Agent-Directed Simulation Symposium, 2005.

[16] M. Scheutz, P. Schermerhorn, Many is more: The utility of simple reactive agents with predictive mechanisms in multiagent object collection tasks, Web Intelligence and Agent Systems (in Press) 3 (1).

[17] J. S. Steinman, Discrete-event simulation and the event horizon, in: PADS ’94:

Proceedings of the eighth workshop on Parallel and distributed simulation, ACM Press, New York, NY, USA, 1994, pp. 39–49.

[18] M. Scheutz, B. Logan, Affective versus deliberative agent control, in: S. Colton (Ed.), Proceedings of the AISB'01 Symposium on Emotion, Cognition and Affective Computing, Society for the Study of Artificial Intelligence and the Simulation of Behaviour, York, 2001, pp. 1–10.

[19] M. Scheutz, P. Schermerhorn, Steps towards a theory of possible trajectories from reactive to deliberative control systems, in: R. Standish (Ed.), Proceedings of the 8th Conference of Artificial Life, MIT Press, 2002.

[20] A. Sloman, B. Logan, Cognition and affect: Architectures and tools, in: Proceedings of the Second International Conference on Autonomous Agents (Agents '98), ACM Press, 1998, pp. 471–472.

[21] T. K. Som, R. G. Sargent, Model structure and load balancing in optimistic parallel discrete event simulation, in: Proceedings of the fourteenth workshop on Parallel and distributed simulation, 2000, pp. 147–154.

[22] L. Wang, S. J. Turner, , F. Wang, Resolving mutually exclusive interactions in agent based distributed simulations, in: Proceedings of the 2004 Winter Simulation Conference, 2004, pp. 783–791.

[23] J. Goic, J. A. Sauter, T. Toth-Fejel, Syncer: Distributed simulations using swarm, in: SwarmFest 2001, Santa Fe, NM, 2001.

[24] N. Minar, R. Burkhart, C. Langton, M. Askenazi, The Swarm simulation system, a toolkit for building multi-agent simulations, http://www.santafe.edu/projects/swarm/overview/overview.html. (1996).
URL `citeseer.nj.nec.com/minar96swarm.html`

[25] L. Gasser, K. Kakugawa, Mace3j: Fast flexible distributed simulation of large, large-grain multi-agent systems, in: Proceedings of AAMAS 2002, 2002, pp. 745–752.

[26] R. Minson, G. Theodoropoulos, Distributing repast agent based simulations with HLA, in: Proceedings of the 2004 European Simulation Interoperability Workshop, Edinburgh, UK, 2004.

[27] N. Collier, RePast: An extensible framework for agent simulation, http://www.econ.iastate.edu/tesfatsi/RepastTutorial.Collier.pdf (2003).

[28] M. Lees, B. Logan, T. Oguara, , G. Theodoropoulos, Simulating agent-based systems with HLA: The case of SIM_AGENT part ii, in: Proceedings of the 2003 European Simulation Interoperability Workshop, Stockholm, Sweden, 2003.

[29] B. Logan, G. Theodoropoulos, The distributed simulation of multi-agent systems, Proceedings of the IEEE 89 (2) (2001) 174–186.