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

Matthias Scheutz and Jack Harris

Abstract In previous work [7], we proposed a general framework for defining agent-based models (ABMs) and introduced 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", which can utilize the additional spatial information to obtain performance improvements. Both algorithms can automatically distribute ABMs over multiple processors and dynamically adjust the degree of parallelization based on available computational resources throughout the simulation runs. However, they are not sensitive to inefficiencies in the sequence in which agents in each parallel simulation instance are updated.

In this chapter, we introduce a minimal framework for describing ABMs and propose various asynchronous scheduling algorithms for agent-based simulations that address the update inefficencies of simulation schedulers. The proposed algorithms work in conjunction with **P-ABM**$_G$ and **P-ABM**$_S$ and allow for efficient simulation runs that can automatically and better utilize the asynchronous nature of parallel distributed agent-based simulations (including split-ups of specific simulation models and dynamic load-balancing). We demonstrate the significant performance gains of the proposed algorithms using an actual agent-based model used for studying female choice and foraging in biological research.

Matthias Scheutz and Jack Harris
Human-Robot Interaction Laboratory
Cognitive Science Program and School of Informatics
Indiana University
Bloomington, IN 47406
e-mail: {mscheutz,jackharr}@indiana.edu

1 Introduction

Simulations of agent-based models (ABMs) have been successfully applied in a variety fields to reveal and elucidate interaction patterns among entities in complex systems that are otherwise difficult to detect and understand. Depending on the investigated problem, the entities – or “agents”[1] – in agent-based models might take a different form. Model varieties range from chemicals or simple cybernetic creatures in artificial life, to web pages, computers, or human users in complex networks, to game-theoretic players in economics, to groups of humans or animals in social studies, biology, and anthropology (see [7] for more references).

What is common to all these diverse models is that they decompose the behavior of complex systems into tractable actions and interactions of individual agents. This is typically achieved by defining *rules* that determine the behavior of individual agents for all possible contexts in which the agent might find itself. A special class of agent-based models, the *spatial* agent-based models, explicitly defines an *environment*, which is typically a metric space. Every agent is situated in a particular location in the environment at any given time, but the location may change over time (e.g., if the agent is moving). Every agent also has an *interaction range* that, given the agent’s location, determines the set of other agents in the environment with which it can interact at any given time; it cannot interact with or have any effects on agents outside of its interaction range.[2]

From a computational perspective, agent-based models are interesting because they often lend themselves to efficient, parallel implementations. One obvious way to parallelize agent-based models, for example, is to spawn a separate computational thread for each agent in a given simulation. These computational processes will take care of computing the agent’s behavior and will of course have to be synchronized to ensure that all agents are updated consistently – we will say more about this shortly. Since for typical models, the number of agents in the model will by far outnumber the cores or processors available on the computer running the model simulations, it might also make sense to distribute simulations across multiple computers to better utilize the intrinsic parallelism in agent-based models. There is, however, a critical difference between distributing simulations over multiple connected computers and parallelizing a given model within one computational process on one computer (e.g., using a parallel programming language like ADA, or a threaded programming language like JAVA). In single process simulations, all agents can access the same environment and can use synchronization mechanisms available within the process, while in distributed simulations the environment has to be replicated on each host computer, and synchronization between agents and environments have to

[1] Agent-based models–sometimes also called “individual-based” models–are often used to simulate the behavior of complex real-world systems. They are used 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).

[2] Note that general agent-based models can be viewed as a special case of spatial agent-based models in that all agents are located in the same location and can interact with all other agents at any given time.

be achieved using networked “inter-process” synchronization primitives. Whereas parallelism within one computational process might be already implemented in the employed simulation environment, dynamic parallelization via distribution of the environment over multiple computers is not available in any of the common modeling environments. Augmenting such environments to support multi-host distribution requires significant programming expertise, which modelers usually do not possess or are unwilling to invest their time, given that their research interest lies in the simulation results and not the computational infrastructure.[3] We believe that modelers should not have to worry about computational issues, but should be able to define their models in their favorite modeling environment, and the computational infrastructure should take care of running these models in the most efficient fashion given the available resources.

We have previously developed algorithms that will support modelers in achieving good turn-around times of model simulations by automatically parallelizing and distributing general and spatial agent-based models [7]. In this chapter, we extend our previous ideas for scheduling agents in simulations of spatial agent-based models by introducing novel scheduling algorithms. These algorithms take advantage of both the inherent parallelism in agent-based models and the interaction ranges of agents in spatial models. We demonstrate that these algorithms can achieve a significant performance improvement over standard scheduling algorithms in the context of our previous parallel and distributed algorithms [7] through a reference implementation in our SWAGES system [8]. This improvement is achieved by virtue of tightly integrating the simulation scheduler with the distribution algorithm. While the proposed algorithms will already be of great utility for modelers, they also pose a variety of interesting open problems for future research, which we will briefly address at the end of our exposition.

2 Distributed Simulations of Agent-Based Models

Various kinds of formalisms and frameworks have been developed to capture this diversity of agent-based models (e.g., some models are essentially physics-based, while others operate solely on a social level). We have previously attempted to define general formal frameworks for hierarchical [6] and spatial [7] agent-based models that were intended to be maximally inclusive. Here, we take the opposite approach and attempt to make due with the smallest formal framework for spatial agent-based models that is sufficient for defining and employing our distribution and scheduling algorithms.

We start with the assumption that each spatial agent-based model (S-ABM) $\mathscr{M}$ has an environment $\langle Env_{\mathscr{M}} \rangle$ that can be modeled as a discrete or continuous *metric* space (e.g., with the Euclidean norm). Such models allow for the simulation of in-

[3] While we don’t have formal evidence for this claim, in our experience modelers are happy to put up with very long single computer simulation runs, before they are willing to entertain the possibility of having to manually distribute their models.

teractions among agents based on a notion of *distance*. Spatial distance is not only 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.), but it is also essential for the parallelization and distribution algorithms we will review in this chapter. We also assume that for each agent A in $\mathscr{M}$ there is a clearly defined *interaction range* I_A that, given the position of all agents in the environment, determines at any given time the set of agents it can (potentially) interact with at that time.[4] Finally, we assume that each model has a set of initial conditions $Init_{\mathscr{M}}$ and a set of terminating conditions $Term_{\mathscr{M}}$ which determine the initial and final states of a simulation of model $\mathscr{M}$ under those conditions. A (discrete-event) simulation of $\mathscr{M}$ is a sequence of updates of all agents A such that for any point in the sequence and any agent, all agents within its interaction range have had the same number of updates since the start of the simulation. The rationale for this definition will become clear later, for now just note that the standard updating sequence in discrete-time simulations falls under this defintion. This sequencing which we call "cycle-based update strategy" updates every agent in A before starting over and updating every agent in A again, and repeatedly looping through the set of agents until a terminating condition is reached.

2.1 A Minimal Framework of Agent-Based Models

We start by defining the notion of spatial agent-based model.

Definition 1 (Spatial agent-based model). A *spatial agent-based model* $\mathscr{M} = \langle Env_{\mathscr{M}}, ATypes_{\mathscr{M}}, Init_{\mathscr{M}}, Term_{\mathscr{M}} \rangle$ consists of an n-dimensional bounded or unbounded metric space $Env_{\mathscr{M}}$ (consisting of locations that can be occupied by agents), a set of agent types $ATypes_{\mathscr{M}}$, a set of initial conditions $Init_{\mathscr{M}}$, and a set of terminating conditions $Term_{\mathscr{M}}$. $Init_{\mathscr{M}}$ is a set of agents and $Term_{\mathscr{M}}$ is a set of functions from the powerset of $Agents_{\mathscr{M}}$ (the set of all possible agents in $\mathscr{M}$ into $\{true, false\}$. An *agent* A is a triple $\langle ID_A, Type_A, State_A \rangle$ which consists of the agent's unique identifier $ID_A \in \mathbb{N}$ (required to be able to dissociate agents that would otherwise be identical with respect to their remaining information), an agent type $Type_A \in ATypes_{\mathscr{M}}$, and an agent state $State_A$, where $State_A = \langle L_A, I_A, T_A, U_A, ... \rangle$ contains the agent's location $L_A \in Env_{\mathscr{M}}$ in the environment, its interaction range $I_A \in Env_{\mathscr{M}}$, a translation function T_A which determines for a given location the maximum distance an agent can travel within one update,[5] an agent update function U_A mapping sets of agent states onto agent states, and any other pertinent information about the agent particular to the model.[6]

[4] Note that the interaction range is allowed to change over time, but at any given point in time it has to be defined and uniquely determinable. Furthermore, note that we are not distinguishing between "sensory" and "actuator" ranges here, see [7] for such a distinction.

[5] We will discuss the reason for this function later.

[6] Note that formally the definitions of "agent-based model", "agent", and "agent state" are co-recursive, i.e., mutually dependent and thus mutually defined. This type of definition requires non-

Equipped with the notion of agent, agent state, and agent-based model we can now define what we mean by a simulation of an agent-based model:

Definition 2 (Simulation of an S-ABM). A *simulation* $S_{\mathscr{M},C_0}$ of an S-ABM $\mathscr{M}$ is defined as a finite sequence of configurations $S_{\mathscr{M},C_0} = \langle C_0, C_1, \ldots, C_k \rangle$ starting with configuration C_0 and ending with C_k. Each configuration C_i ($0 \leq i \leq k$) is a set of agents of some type in $ATypes_{\mathscr{M}}$ – we will use $Agents_{\mathscr{M}}$ to denote the set of all possible agents supported by $\mathscr{M}$. C_0 is an initial configuration in $Init_{\mathscr{M}}$. C_k is a terminal condition such that for some function $f \in Term_{\mathscr{M}}$ $f(C_k) = true$ and there is no configuration C_j with $j < k$ and $f \in Term_{\mathscr{M}}$ such that $f(C_j) = true$. We also require for all configurations C_i and C_{i+1} ($0 \leq i < k$) to be "consistent", i.e., if C_j "follows" C_i in the simulation sequence (i.e., $j = i + 1$) then C_j is obtained from C_i by (simultaneously) updating a subset of agents $\mathscr{A} \subseteq C_i$ such that all agents $A' \in C_i$ whose interaction range intersect with that of some $A \in \mathscr{A}$ are (already) in $\mathscr{A}$. An agent $A \in C_i$ (part of a simulation $\langle C_0, C_1, \ldots, C_k \rangle$ of S-ABM $\mathscr{M}$) is said to *be at the n-th cycle* if A has been updated n times, i.e., $U_A^n(State_A^0) = State_A$ where $State_A^0$ was the state of A in C_0 and $State_A^n$ is its state after n updates. Finally, a configuration C_i is said to be *at cycle n* if all of its agents are at the n-th cycle.

Note that simulations of agent-based models are, by definition, *consistent sequences* of configurations where all subsets of agents with intersecting interaction ranges are at the same cycle (this is a more permissive notion of configuration than the one implicitly underlying typical event-based simulations, namely that all agents in a configuration must be at the same cycle). Consequently, sequences of configurations that are *not consistent* (i.e., where agents get updated in an "inconsistent fashion" as would be the case if an agent that was updated twice already got updated based on its interaction with an agent that had only been updated once) are not simulations of agent-based models. However, the above definition of "consistent configuration" does not require that there be a "unique successor" of a given configuration (as is typically defined for discrete-event-based simulations), because for any given set of agents there could be many possible configurations that follow. Consequently, an initial configuration will give rise to a directed graph of configurations, call it the "configuration graph of $\mathscr{M}$", which could be infinite (e.g., if the sequence contains only non-final configurations of updates of the same agent that does not change its state and update thus never lead to a final configuration) – we will use $Cfg_{\mathscr{M}}$ to denote the graph of all configurations of $\mathscr{M}$ that successively follow any configuration from $Init_{\mathscr{M}}$.

While we are, in general, interested in the shortest path through the $Cfg_{\mathscr{M}}$ since that path will give us the desired result (i.e., the terminal configuration(s) we are interested in), the shortest path may not be unique. And, moreover, it is possible that there are two shortest paths that result in *different terminal conditions*. Standard discrete-time simulations do not usually distinguish between different terminal states that differ only with respect to the ordering of the agents in the "cycle-based

well-founded set theory as a formal framework, where the "Solution Lemma" ensures that these kinds of structures are properly defined and exist [1].

update strategy" (i.e., there might be two agents in the same update cycle that cause the termination of a simulation), although this problem can be easily avoided by finishing the updates of all agents within the same cycle (e.g., by requiring as an additional condition for termination that all agents *are* at the same cycle). We will now formally define the notion of update strategy:

Definition 3 (Update strategy). An *update strategy* or *update policy* for an S-ABM $\mathscr{M}$ is a mapping $\pi_{\mathscr{M}} : Cfg_{\mathscr{M}} \mapsto \mathscr{P}(Agents_{\mathscr{M}})$ from possible (consistent) configurations to the powerset of all agents, which effectively in each possible configuration selects a subset of agents for updating. An update strategy $\pi_{\mathscr{M}}$ is *consistent* if for all configurations $C \in Cfg_{\mathscr{M}}$ with $\pi_{\mathscr{M}}(C) = \mathscr{A}$ and all $A \in \mathscr{A}$ at cycle k, there is no agent $A' \in C$ such that A' within I_A and A' is at a different cycle $l \neq k$. A simulation $\langle C_0, C_1, \ldots, C_k \rangle$ is updated based on a *cycle-based update strategy* whenever for two configurations in sequence in the simulation all agent states are at most one cycle apart (i.e., for any two configurations in sequence C_l and C_m with $A_l \in C_l$ and $A_m \in C_m$, if A_l is at the $i-th$ cycle and A_m is at the $j-th$ cycle, then $|i-j| \leq 1$).

It follows immediately that cycle-based update strategies (such as updating agents based on some ordering of their unique IDs) are consistent. While cycle-based update strategies are commonly used in and appropriate for discrete-event simulations on single computer systems, they do not necessarily give rise to good performance in distributed simulations, as we will see in Section 3.

As a side remark, simulations of agent-based models, as defined above, are *deterministic* and thus *reproducible* from initial configurations. However, it is sometimes desirable to allow for "non-deterministic" state transitions (e.g., to model probabilistic state transitions where each transition has a certain likelihood associated with it). The above definitions can be straightforwardly augmented to allow for non-determinism by dropping the requirement that agent updates be functions and constructing them as annotated relations instead, 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.[7]

2.2 Distributing Simulations of Agent-Based Models

Spatial agent-based simulation models can be automatically parallelized and distributed in different ways. One obvious way is to run each agent on its own processor. Before an agent can update its state, it needs to collect the current state information from all other agents (running on other processors). Once the information is

[7] 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).

available, the agent updates its state and begins the update cycle again. All processors update their respective agents in the very same cycle-based fashion to ensure the correctness of the results. Another possibility is to determine in advance whether an agent needs the state of another agent for its update and to distribute agents based on these dependencies (e.g., subsets of mutually dependent agents end up on the same processor, which limits the exchange of state information to exchanges among local agents). Other options are to predict or (empirically) determine the actual update time of an agent and run computationally expensive agents on separate processors, while running computationally cheap agents together on one processor.

The ideal case would be a setup where only one partition $\Pi(C_0)$ of the agents in the initial configuration C_0 has to be computed and after distributing and initializing all agents on their respective processors, each processor can update its agents independently and asynchronously until a final configuration is reached.[8] Unfortunately, this case is rarely true of agent-based models given that they are typically used for the study of interactions among agents. Hence, additional mechanisms are required to synchronize the states of agents residing in different simulations on different processors. "Synchronization" here means that if a simulation instance running on processor P_i requires the state of an agent E_j from a "remote" simulation instance running on another processor P_j, then the simulation on processor P_j needs to be able to send this information back to the simulation on processor P_i.

Which of the above approaches works best will depend on various factors, including the complexity of the update function of the involved agents, the distribution of agent types in a particular setup, the computational overhead of sending state information requests and receiving them (including network latencies), the pool of available processors (e.g., individual speeds, etc.) and whether this pool 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.

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

Definition 4 (Split of Configuration). Let $\mathscr{M}$ be a S-ABM, C a configuration in $Cfg_{\mathscr{M}}$, and $Proc = \{P_1, P_2, \ldots, P_n\}$ a set of available processors ("processor pool"). Then a *split* P^C_{Proc} of C is a mapping $P : C \mapsto Proc$–called *agent-processor assignment*–of agents to processors P_i in $Proc$.

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

Corollary 1. *A split P^C_{Proc} induces a partitioning Π_C of a configuration C into i disjoint subsets of agents Π_{C_i} in C.*

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

[8] Note that detecting final configurations in a distributed simulation can be very tricky and will be briefly addressed in the Discussion section.

Each “subconfiguration” 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 agents on each processor P_i independently as long as the states of agents updated by P_i do not depend on the states of agents updated by other processors P_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 agents 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 sufficient condition for detecting it in Section 2.3.

Definition 5 (Update Independence). An agent A_1 is *update-independent* $UI_C(A_1,A_2)$ of another agent A_2 in a configuration C (with $A_1,A_2 \in C$), if the updated state of A_1 in each following configuration C' of C is the same as in each respective following configuration $(C-A_2)'$ of $C-A_2$ (i.e., the configuration obtained from C by removing agent A_2). A_1 is called *update-dependent* on A_2 in C if $\neg UI_C(A_1,A_2)$. A_1 and A_2 are called *mutually update-independent* in C if A_1 is update-independent of A_2 and vice versa (see Figure 1). Two subconfigurations $C_1,C_2 \subseteq C$ are *mutually update-independent* $UI_C(C_1,C_2)$ if $\forall A_1 \in C_1, A_2 \in C_2 [UI_{C_1}(A_1,A_2) \wedge UI_{C_2}(A_2,A_1)]$. A set of configurations $\mathscr{C}$ is *update-independent* if $\forall C_1,C_2 \in \mathscr{C} UI_C(C_1,C_2)$. 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 agent A_i cannot have any effect on A if its removal does not change the update of A. Note that update-independence is *not symmetric* (that is why we need the additional notion of “mutual update-independence”): it is possible that one agent A_1 is update-independent in C from another agent A_2, while the latter is not update-independent in C from the former (e.g., consider A_1 with interaction range of 10 located in (0,0) and A_2 located in (0,50) with interaction range 100 for its sensors only; then A_2 can sense A_1 and might change its behavior based on the perception without being able to affect A_1, while A_1 is oblivious to A_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”:

Corollary 2. *Let C be a configuration and Π_{C_i} update-independent configurations obtained by splitting C via P^C_{Proc}. Then $update(C) = \bigcup update(\Pi_{C_i})$ (where $update()$ is applied to all agents in the configuration).*

Proof. By induction on the size of the split. The base case, $C = P^C_{Proc}$ is obvious. Assume the Corollary has been shown for splits of size n. Then observe that for splits of

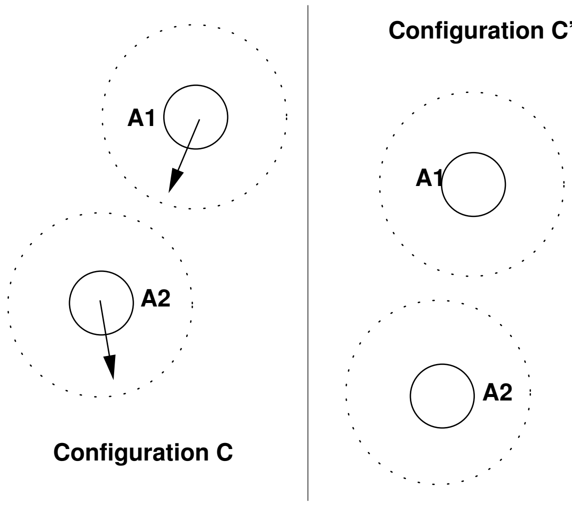


Fig. 1 An illustration of "update independence". Two agents A_1 and A_2 are both about to move in different directions (as indicated by arrows) in a configuration C. Since their interaction ranges (indicated by dashed circles) within which they can affect their environment do not overlap, either agent 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, A_1 and A_2 are mutually update-independent.

size $n+1$, $update(C) = update(P^C_{Proc}) = update(\bigcup \Pi_{C_i}) = update(\bigcup^{-(n+1)} \Pi_{C_i} \cup \Pi_{C_{n+1}})$ where $\bigcup^{-(n+1)}$ is the union over the first n configurations. By induction assumption, it follows that $update(\bigcup^{-(n+1)})$ is the same as $\bigcup update(\Pi_{C-(n+1)})$, the union of the updates of all configurations Π_{C_i} except for $\Pi_{C_{n+1}}$. Now observe that $update(\Pi_{C-(n+1)} \cup \Pi_{C_{n+1}}) = update(\Pi_{C-(n+1)}) \cup update(\Pi_{C_{n+1}})$ given that the update of an agent $A \in \Pi_{C-(n+1)}$ does not depend on any agent in $\Pi_{C_{n+1}}$ since A is update-independent from all agents in $\Pi_{C_{n+1}}$ (by def. of update-independence of two configurations). The analogous argument shows that is is also true for all agents in $\Pi_{C_{n+1}}$. Hence, $update(C) = \bigcup update(\Pi_{C_i})$.

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 ($C_0$,$Proc$,$\mathcal{M}$) $C := C_0$
while $\neg\exists f \in Term_{\mathcal{M}} : f(C) = true$ do
    compute an update-independent split $P^C_{Proc}$ of $C$ for $Proc$
    distribute each subconfiguration $\Pi_{C_i}$ onto $P_i$ in $Proc$
    compute $update(\Pi_{C_i})$ on each $P_i$ and merge all $\Pi_{C_i}$ into $C$
    $Proc := update(Proc)$
end while
```

It is a direct consequence of merging at the end of each update that the algorithm[9] is "step-wise correct" in the following sense:

Definition 6 (Step-wise Correctness). Let $\mathscr{A}$ be a parallel algorithm for updating a spatial agent-based simulation $S_{\mathscr{M},C_0} = \langle C_0, C_1, C_2, \ldots, C_{final}\rangle$ of an S-ABM $\mathscr{M}$. $\mathscr{A}$ is *stepwise correct* if it produces a sequence of split configurations $\langle \Pi_{C_0}, \Pi_{C_1}, \Pi_{C_2}, \ldots, \Pi_{C_{final}}\rangle$ such that $C_k = \bigcup(\Pi_{C_k})$ for all $0 \leq k \leq final$.

Corollary 3. P-ABM$_G$ *is step-wise correct.*

Note that the above algorithm is *adaptive* because the set of available processors is updated after every configuration update. Hence the algorithm can take the new set of resources (e.g., a larger number of available processors) 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 agents. At the same time, if updating an agent is very expensive, splitting agents 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 agents, this configuration will be updated on one processor and thus incurs a computation cost linear in $|C_i|$, which is in the worst case $\mathscr{O}(|C|)$. In such a case it is likely better to further split agents in $|C_i|$, distribute them over different processors, and use a mechanism to request and transfer the states of update-dependent agents in other subconfigurations as part of the update of an agent on-demand.

2.3 Towards Exploiting Properties of Spatial ABMs

As already mentioned, the important missing ingredient that is needed 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. In order to determine whether a split is update-independent for a given pool of processors *Proc* would require repeated computation of a split, independent update of all subconfigurations, and then a comparison of the merged updated subconfigurations to the update of the whole configuration. This means that the computational cost (in terms of space and time)

[9] Note that **P-ABM**$_G$ is an *algorithm* because it is always possible to compute a (trivial) update-independent split in the following inefficient 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 processor): 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.

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 agents: being within each others' interaction range. For, clearly, agents that are *not* within interaction range of each other in a given configuration cannot possibly have any effect on each other, by definition, and are thus mutually update-independent.

Note that being outside of each other's interaction range is a "conservative" estimate for mutual update independence, because two agents can still be update-independent even if they can sense each other (either because they do not take 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 agent always ignores perceptions of its own kind). The general difficulty connected to any better derivation of potential interactions, however, is how to determine them automatically from the agent update functions, which may not be possible in a practical implementation if their representations are not explicitly accessible (and even then, this will, in general, only be possible in a limited way).

Since each agent A, as part of its state, contains a translation function T_A which determines for a given location the maximum distance the agent can travel within one update cycle, the set of locations that A can influence after k of its update cycles is given in terms of T_A^k (i.e., applying T_A repeatedly up to k times to each location in the set of locations returned after each application). In a continuous 2D metric environment with $T_A > 0$, this will be the radius of an expanding circular region (as T_A^k amounts to all locations within $k \cdot T_A$). Call this expanding subspace of the environment that results from the motion of an agent starting in a given configuration C_i the agent's "event horizon":[10]

Definition 7 (Event Horizon). The *event horizon* $EH(A, C_i, k)$ of an agent A starting in configuration C_i is the set of all locations T_A^k after k updates based on its location in C_i.

Clearly, the event horizon of agents A in metric environments with $T_A > 0$ is monotonically increasing, symmetric, reflexive, but not transitive (which is important for computing dependencies among agents). Figure 2 shows the expanding event horizon in a metric 2D environment where T_A models the "maximum speed of locomotion" of A.

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

[10] The term *event horizon* has been previously used in a slightly different sense in the domain of parallel simulation. E.g., "event horizon" in [11] refers to the set of events $\mathscr{E}$ that can occur before the first consequent event E' generated by an event $E \in \mathscr{E}$. Hence, it is the set of events $\mathscr{E}$ 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*.

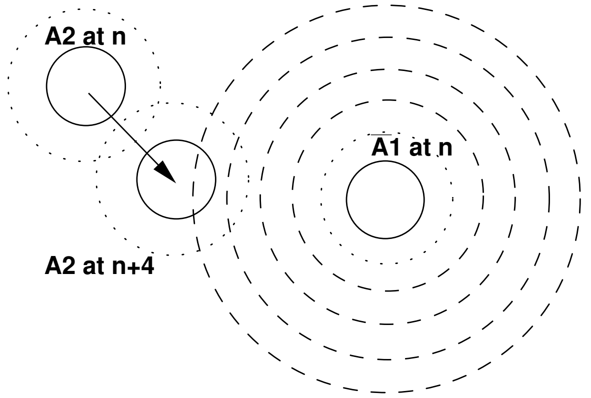


Fig. 2 An illustration of the event horizon. Agent A_2 moves from its position at cycle n to the new position at cycle $n+4$ (indicated by the arrow). The position of the agent represented by proxy agent $\overline{A_1}$ at cycle n is known, but not thereafter. The dashed circles indicate the increasing event horizon of that agent for subsequent cycles (including the maximum of the two sensory ranges–sensory ranges are indicated by dotted circles). At cycle $n+4$ A_2 intersects with the event horizon of $\overline{A_1}$ indicating that the actual position of A_1 is required before the update of A_2 can be computed.

merged and which can continue, we introduce the notion of a "proxy agent", which serves as a (local) placeholder in a subconfiguration for the last known state of an agent updated in another subconfiguration (on another processor).

Definition 8 (Proxy agent). A *proxy agent* $\overline{A}$ of an agent A (in the following always denoted by a bar) consists of the agent's state with the location L_A replaced by a set of possible locations $\overline{L_A}$ and the update function U_A replaced by $\overline{U_A}$ (the function that just repeatedly applies $T(A)$ to $\overline{L_A}$ on each cycle).[11]

Proxy agents merely have a *representational function* and cannot be updated like regular non-proxy agents (i.e., they cannot change their state across configurations). Yet, they are used to compute the event horizon of the agent in subsequent configurations based on the last known configuration at which the proxy agent was updated by repeatedly applying $\overline{U_A}^k$ to each $L \in \overline{L_A}$. That way, given the state of a proxy-agent $\overline{A_j}$ (representing agent A_j in subconfiguration C_j) it is is possible to determine the subspace of the environment on which an agent A_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 agents in C_i) before any interaction between A_j and any $A_i \in C_i$ is possible.

We can now state an important lemma (for a proof, see [7]):

Lemma 1 (Interaction Lemma). *Let C_1 and C_2 be two subconfigurations of a configuration C containing only non-proxy agents and let C_1^* and C_2^* be the configurations obtained from C_1 and C_2 by adding the proxy agents in $\overline{Agents_2}$ and*

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

$\overline{Agents_1}$ that represent the states of some non-proxy agents in C_2 and C_1, respectively. Moreover, let n be the largest number such that no non-proxy agent $A_1 \in C_1$ has $L_{A_1} \in EH(\overline{A_2}, C_1, n)$ for any $\overline{A_2} \in \overline{Ent_2}$ and no non-proxy agent $A_2 \in C_2$ has $L_{A_2} \in EH(\overline{A_1}, C_2, n)$ for any $\overline{A_1} \in \overline{Ent_1}$. Then for all $k \leq n$, $U^n_{\mathscr{M}}(C_1) \cup U^n_{\mathscr{M}}(C_2) = U^n_{\mathscr{M}}(C_1 \cup C_2)$. Or put differently, C_1 and C_2 are mutually update-independent for at least the first n updates.

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 agents in either configuration contains a location of a non-proxy agent in that configuration. When such a configuration is reached, the actual state of the agent represented by the proxy agent needs to be obtained. Hence, we can formulate the following refined version of **P-ABM**$_G$ for S-ABMs:

```
**P-ABM**$_S$ ($C_0$,$Proc$,$\mathscr{M}$)
$oldProc := \emptyset$
$k := 0$
**while** $\neg\exists f \in Term_{\mathscr{M}} : f(C_k) = true$ **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 $P_i$ 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 $EH(\overline{\Pi_{C_j}}, C, k)$ for the last known state from some configuration $C$
  **for** proxy agent $A_j$ that has a non-proxy agent $A$ within $EH(\overline{\Pi_{C_j}}, C_k, k)$ **do**
    get state of $A_j$ at $k$ from $processor_j$ and update $\overline{E_j}$
  **end for**
  compute $(\Pi^*_{C_{k,i}})$' := $update(\Pi^*_{C_{k,i}})$ on each $processor_i$
  update $Proc$
  **if** $oldProc \neq Proc$ **then**
    merge all $C_{k+1} := \bigcup U_{\mathscr{M}}(\Pi_{C_{k,i}})$
  **end if**
  $k := k+1$
**end while**
```

It follows that **P-ABM**$_S$ is step-wise correct (see [7] for a proof sketch).

3 Update Strategies for Distributed Parallel Agent-Based Simulations

The main advantage of **P-ABM**$_S$ over **P-ABM**$_G$ is that it does not require all simulation instances running on different processors to synchronize after all agents in each of the distributed simulation instances have been updated once. Rather, as long as all non-proxy agents located in a given simulation instance are located outside the event horizon of all proxy agents, the simulation instance can update its agents without requiring information from any of the other simulation instances. On the other hand, when there are potential interactions, as determined by the proxy agents' event horizons, simulations do not necessarily have to be merged, instead it suffices to update only the proxy agents based on the communicated locations of the non-local agents (in other simulation instances) they represent. Consequently, it is also not necessary to compute new update-independent splits before every update cycle (although simulations will still have to be merged and splits will still have to be recomputed, as with **P-ABM**$_G$, should the processor pool *Proc* change to preserve the adaptiveness in **P-ABM**$_S$). To elucidate how this asynchronous update could work, we start with an intuitive example, and then look at the properties of asynchronous updates more formally.

Suppose $agent_{15,4}$, i.e., the agent with $ID = 15$ in simulation instance 4, requires at its cycle 321 an update for its proxy agent $proxy\_agent_{64,7}$ (i.e., the proxy agent representing the agent with $ID = 64$ in simulation instance 7). Furthermore, let's assume that both simulation instances, 4 and 7, have been running asynchronously up to that point without communicating with each other. When simulation instance 7 gets the request from simulation instance 4 to send the pertinent information (i.e., the reduced state) of $agent_{64,7}$, $agent_{64,7}$ is already at cycle 598 (in simulation instance 7 due to the asynchronous updates). At first glance, the mismatch in cycle numbers seems to prevent an information transfer that can be used in a way that will keep the distributed simulation consistent. On further examination, however, it turns out that it is completely unproblematic for simulation instance 7 to communicate the current reduced state of $agent_{64,7}$ and for simulation instance 4 to use it (instead of the state of $agent_{64,7}$ at cycle 321) – why is that? The answer lies in the event horizon of proxy agent $proxy\_agent_{15,4}$ in simulation instance 7, which represents $agent_{15,4}$ from simulation instance 4: if there had been any chance for $agent_{15,4}$ to interact with agent $agent_{64,7}$ before cycle 598, then $agent_{64,7}$ would have ended up being located within the event horizon of $proxy\_agent_{15,4}$ in simulation instance 7 before that cycle and simulation instance 7 would have requested an update (i.e., reduced state) from $agent_{15,4}$ in stimulation instance 4. However, since no such request occurred based on our assumption, $agent_{64,7}$ never ended up being located within the event horizon of $proxy\_agent_{15,4}$ and therefore never had a chance to interact with $agent_{15,4}$, at least until cycle 598. Since there cannot be any earlier interaction, simulation instance 4 can simply use the reduced state of $agent_{64,7}$ at cycle 598 and set its proxy agent $proxy\_agent_{64,7}$ to that state, and even skip updating the agent's event horizon until all other agents reach cycle 598.

We formally summarize the above argument in a proposition:

Proposition 1. *Let $A_{i,m}$ and $A_{j,n}$ be agents with agent IDs i and j, respectively, and let S_m and S_n be two simulation instances with $A_{i,m} \in S_m$ at cycle m and $A_{j,n} \in S_n$ at cycle n, with $m < n$ such that for all cycles $m \leq k \leq n$ $A_{j,n}$, is not in the event horizon of $\overline{A_{i,k}}$ (where $\overline{A_{i,k}}$ is the proxy of agent $A_{i,k}$ in S_n). Then for all cycles $m \leq k \leq n$, $A_{i,k}$ is not in the interaction range of $A_{j,k}$ and vice versa.*

Proof. Suppose there is a cycle l such that $m \leq l < n$ at which both agents are within interaction range and suppose further that cycle $c \leq m$ was the last time that simulation instance S_n updated its proxy agent $\overline{A_{i,l}}$ based on the actual state of $A_{i,l}$.

Since the event horizon of is the maximum range at any given cycle within which an agent can interact and for no cycle c with $j \leq c \leq n$ was $A_{j,c}$ within the event horizon of $\overline{A_{i,c}}$, by deviation of $\overline{A_{i,c}}$, $A_{i,c}$ could not have interacted with $A_{j,c}$. Contradiction.

Intuitively, it seems clear that the ability of simulation instances to run asynchronously and only communicate agent states when necessary should lead to performance improvements, and we have indeed been able to show previously that running simulations asynchronously using the above proposition leads to better performance than running simulations in lock-step (as is required for **P-ABM**$_G$) [7]. However, the extent of the performance improvement depends on several factors, including the complexity of the agent update function and the distribution of the agents across simulation environments, but most importantly on the update strategy *given agent update functions and distributions*. Unfortunately, there is no *general update strategy* that will yield optimal results, i.e., maximum parallelism among distributed simulation instances.

To see this, consider two simulation instances with two agents each as arranged in the left part of Figure 3. Agents A and D are non-proxy agents in simulation instance 1 and proxy agents in simulation instance 2, and, conversely, agents B and C are non-proxy agents in simulation instance 2 and proxy agents in simulation instance 1. Each agent only moves in the direction indicated by the arrow pointing away from its center. The dashed circles indicate the agents' interaction ranges. We further assume that the maximum change in location that agents can perform in one update cycle is large enough so that agent D will be in the event horizon of proxy agent B in simulation instance 1 and agent C will be in the event horizon of proxy agent A in simulation instance 2, hence requiring both simulation instances to get the updated states of the non-proxy agents from the other simulation instance. In fact, an update strategy that decides to update agents C and D first, can lead to a lock-step process where on every cycle updates for proxy agents are required, which will, in turn, trigger updates for the remaining agents. As a result, a "cycle-based update strategy" (which is the default in many simulation environments) is not a good choice for the given scenario because it requires updates on every cycle and forces both simulation instances to be in sync at each cycle, thus effectively forcing the distributed simulation to run in lock step.

If, on the other hand, simulation instance 1 updated agent A and simulation instance 2 updated agent B a few times before updating agent C, then the communicated state information would show that agent A has moved out of the way and that agent C could move freely for a certain number of cycles until it has the same cycle number as proxy agent A. In fact, the best update strategy for a situation where a simulation has to be run for a fixed number of cycles n is to first update agents A and B for n cycles and then update agents C and D for n cycles. This is possible because agent A will never be in the interaction range of any agent in simulation instance 1, and agent B will never be in the interaction range of any agent in simulation instance 2. Hence, they are update-independent and can be updated until the terminating condition is reached. As a result, this update strategy will require only one communication of the updated states of A and B (namely after the first update of C and D), then C and D too can be run to completion for $n-1$ cycles – note that at least one update of state information is necessary given the initial condition, hence this update strategy is optimal.

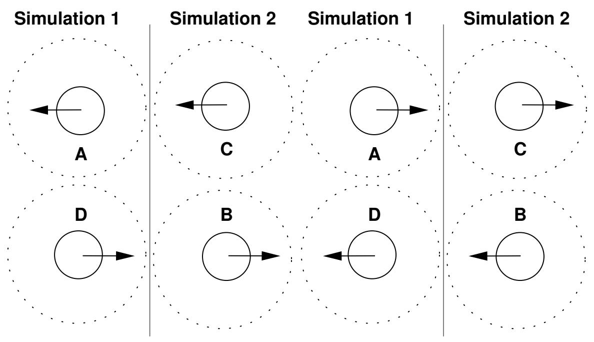


Fig. 3 Two simple scenarios demonstrating that a general optimal update strategy does not exist (see text for details).

While is it possible to have optimal update strategies for particular scenarios such as the above, it is now also easy to see that there cannot be a general optimal update strategy that is best for in every scenario. Consider the right part of Figure 3, which is the same setup as on the left except that all agents move in opposite directions. Hence, the optimal update strategy is to "reverse" the update sequence from before, starting with updating agents C and D for n cycles, followed by updating A and B for n cycles (with one state update required for each proxy agent after cycle 1). Since both scenarios are the same – same number of agents, same split – yet the optimal strategy is different for each scenario, there cannot be a general algorithm that determines the optimal strategy based on the number of agents and splits alone. Rather, the direction of movement, which is determined by the agent update function, is essential for selecting the best update strategy, hence:

Fact 1 *There is no general algorithm which for any given split of a spatial agent-based model simulation can determine the best update strategies for each simulation instance (without knowledge of the specific agent update function).*

Even though there is no general algorithm to determine optimal update strategies, it still makes sense to attempt to define heuristics that will improve over the above lock-step behavior caused by the "cycle-based update strategy". In the following, we will discuss four proposals of such strategies and later show how three of them can be an improvement over the "cycle-based update strategy" in an agent-based model taken from a real-world modeling application.

3.1 A General Alternative Scheduling Strategy for Agent-Based Models

Alternative scheduling strategies for ABMs can be used to avoid some of the inefficiencies associated with "cycle-based update strategies". Unlike the typical "cycle-based update strategy", which require repeatedly updating every agent in a given configuration once (without updating one agent twice unless all other agents have been updated at least once), alternative scheduling strategies relax this update constraint. Instead, they allow for agents to be at different cycles within one simulation instance as long as these cycle differences do not lead to inconsistent update sequences.

In a way, alternative scheduling strategies apply the idea of **P-ABM**$_S$, that parallel simulation instances can run asynchronously as long as none of their agents are within each others' event horizon, at the level of an individual simulation instance: all agents in a subset $AS \subseteq C$ of a configuration C can be updated until one of the agents $A \in AS$ enters the event horizon of some agent $B \notin AS$. Note that for this idea to work, we have to extend our notion of proxy agent to agents *within the same simulation instance*. Specifically, we have to first select a subset of agents $AS = \pi(C)$ based on our selection policy π given the current configuration C such that all agents $A \in AS$ are at the same cycle n. Then we replace the remaining set of agents $RS := C - AS$ with proxy agents that will get updated along with the selected non-proxy agents. This is done to detect possible interactions with agents outside of AS at which point AS is no longer update-independent. To make this update strategy work in a consistent fashion, it is important to pay attention to the cycle at which each agent $B \in RS$ is when their respective proxy agent is initialized: if B is at a cycle $\leq n$, then the event horizon of its proxy agent $\overline{B}$ has to be computed for cycle n; otherwise the proxy agent will not be updated until its cycle number $> n$ is reached. We can summarize this scheme as the general alternative scheduling algorithm **AltSched**$_G$:

```
**AltSched**$_G$ $(C_0, \pi, \mathscr{M})$
$C := C_0$
**while** $\neg\exists f \in Term_{\mathscr{M}} : f(C) = true$ **do**
    $AS := \pi(C)$ (with all $A \in AS$ at the same cycle $k$)
    $RS := C - AS$ (*)
    $\overline{RS} := \{\overline{B} | B \in RS\}$
    $C := (C - RS) \cup \overline{RS}$
    **while** $\neg\exists A \in AS, B \in RS : L_A \in EH(B, C, cyc(A)) \wedge \neg\exists f \in Term_{\mathscr{M}} : f(C) = true \wedge cycle(AS) < k + maxupdate$ **do**
        $C := update(AS \cup \overline{RS})$
    **end while**
    $C := (C - \overline{RS}) \cup RS$ (**)
**end while**
```

There are several important points to notice about **AltSched**$_G$. First note that **AltSched**$_G$ is consistent (which follows from the Interaction Lemma) but will in general not lead to the same simulations as cycle-based strategies. While the latter are guaranteed to find a termination condition (if it exists) given that they perform a breath-first search, whether **AltSched**$_G$ will find a terminal condition critically depends on how the update policy π selects subsets of agents and on how long they are updated using $maxupdate$ (the maximum number of updates to be performed on a set AS before another set is chosen again based on π). In infinite configuration graphs, for example, a depth-first update strategy might fail to find a terminating condition. A simple solution to this problem is to require of π that no agents in a configuration C be selected that are more than a maximum difference δ cycles ahead of any other agent in C.

Second, **AltSched**$_G$ subsumes the standard "cycle-based update strategy" using $maxupdate = 1$ and the policy $\pi(C) = C$ for all C in a model $\mathscr{M}$.

Third, with only minor modifications it lends itself to multiple asynchronous parallel runs: simply recursively apply π to RS at the line marked (*) yielding a set of agents AS_i and their associated proxy agents RS_i until $RS_i = \emptyset$, and then merge all updated non-proxy agent sets AS_i to obtain the new configuration (instead of replacing the proxy agents $\overline{RS}$ with their non-updated counter parts RS in the sequential version). This way of parallelizing a single simulation instance might be preferable over the non-parallelized version even though the parallelization incurs a small computational overhead because it will be able to automatically utilize real parallelism available on multi-processor and multi-core machines as well as idle processor time on a single processor with only one core (e.g., due to wait times on network communication in the context of **P-ABM**$_S$, or various OS blocking).

Fourth, by being able to change the update sequence of agents, using **AltSched**$_G$ can lead to much shorter simulation runs if the update policy π is sensitive to terminal conditions. For example, suppose in a simulation with 1000 agents the goal is for at least one agent to reach a particular goal location in the environment and the terminal condition is thus defined by one of the agents being in that location. Moreover, suppose that the update policy π gives priority to agents that are close to the goal location and that in the initial configuration a group of 10 agents which is outside of the interaction range of other agents is headed directly towards the goal location. To keep things simple, let us also assume that all agents travel at the same speed. Repeatedly selecting these 10 agents for update will then cause the simulation to reach a terminal state without ever having to update any of the other agents (because they will never be in the event horizon of any of the other 990 agents). As a result, the run time under the "goal-sensitive" policy π is about 1% of the runtime of the policy corresponding to the "cycle-based update strategy".

And finally, **AltSched**$_G$ can be combined with **P-ABM**$_S$ and should lead to a performance improvement for distributed simulations if general update policies can be defined that they are sensitive to the update requirements of distributed simulation instances, which we will address next.

3.2 Combining **AltSched**$_G$ *and* **P-ABM**$_S$

The goal of combining **AltSched**$_G$ and **P-ABM**$_S$ is to significantly reduce the overall runtime of parallel distributed simulations compared to the standard "cycle-based update strategy". Hence, we need to define general policies for **AltSched**$_G$ that will select agent groups for updates in a way that better exploits the parallelism of distributed simulations in **P-ABM**$_S$. One of the main performance reducing factors that we have seen in our previous implementation of **P-ABM**$_S$ (with the cycle-based update strategy run by all simulation schedulers) is that simulation instances are *blocked*, i.e., that they cannot update any of their agents without first having obtained updated information on one of their proxy agents. Blocked simulations lead to idle processor time and cannot continue to utilize their computational resources until they get the requested updates. Hence, time wasted due to blocking can be reduced by any policy that is able to anticipate blocks in a remote simulation instance and update its local agents in such a way that the update information is available when requested by the remote simulation instance. Note, however, that giving preference to agent groups that will reduce remote blocking can be in tension with the goal of selecting local agent groups (within each simulation instance) that are likely to make the most progress towards reaching a terminal condition – while we briefly return to this problem in the Discussion section, we will focus here on how we can reduce the latencies and delays in running distributed simulations introduced by remote blocks.

While it is in general not possible for a given simulation instance to detect whether and when a remote simulation will require information about one of its local agents, it is possible to compute conservative estimates that if executed by all simulation instances will improve overall system performance. For example, we can determine for each local agent the earliest cycle at which the local agent could be in the event horizon of some proxy (i.e., remote) agent. An update strategy could then decide to give preference to updates of those agents, which will cause the local simulation instance to block earlier than it otherwise would have had it updated other agents first. The benefit of such "early blocks" is that the remote simulation instance can get the updated state from the blocked agent as part of the blocked simulation's request for an update on the proxy agent. Since the remote simulation instance uses the same update policy and thus also gives preference to agents that are likely to need update information in the near future, it is probable that it will either already or at least soon have an update available. Hence, if all simulation instances give preference to updates of agents whose state information will be requested by remote simulation instances in the future, the overall effect is that cycles with blocking agents will occur more frequently in the beginning of a simulation sequence compared to the standard cycle-based updates. As a result, simulation instances will likely still be able to update some of their (non-blocked) local agents while they are waiting for state updates for blocked agents as opposed to a simulation instance waiting to update any agents until the state information is received for all blocked agents. In sum, policies that are sensitive to the information demands of remote simulation instances will in many cases be able to reduce the idle time of parallel simulation instances, which in turn will lead to overall shorter simulation runs (everything else being equal).

We now define four general policies that are intended to reduce the overhead associated with blocking simulation instances in a distributed simulation based on **P-ABM**$_S$.

Remote Event First.

The *Remote Event First* policy projects out the next potential "event" with an agent on a remote host (i.e., a local agent ending up in the event horizon of a remote agent). *Remote Event First* intuitively has benefits because it increases the number of agents that can be run at any one time in every simulation instance and reduces the risk that the simulation instance is completely blocked waiting for state updates from remote simulation instances. This method of ordering performs a depth-first traversal through the configuration graph and therefore is not guaranteed to terminate. Variations of *Remote Event First* (as mentioned above) can be implemented utilizing mechanisms that disallow an agent to advance too far into the future without catching up other younger agents in the same simulation instance.

Remote Blocks Then Remote Event First.

The *Remote Blocks Then Remote Event First* policy is a cooperative variation of the *Remote Event First* policy that works with the other nodes to identify which agents to run next. This policy can drastically improve performance because it will quickly unblock a remote agent that is traversing though the simulation timeline. When an agent becomes blocked, that simulation instance shares this information with all of the other simulation instances causing them to give immediate priority to those agents whose updates will allow the blocked agent to progress. If some of the selected agents are also blocked, a simulation instance will revert back to the *Remote Event First* criteria.

Youngest First.

A *Youngest First* policy simply chooses the agent with the lowest cycle time from all of the potentially update independent agents. This method of ordering has a breadth-first type of traversal through the configuration graph. The main benefit of such a selection policy is that it is guaranteed to terminate if an exit criterion is reachable.

Remote Blocks Then Youngest First.

The *Remote Blocks Then Youngest First policy* is a variation of the *Youngest First* policy that cooperatively works with the other nodes to identify which agents to run next as described above. As with the *Remote Blocks Then Remote Event First* policy, it will revert to its base strategy of *Youngest First* when it cannot advance agents whose updates are requested by remote simulation instances because they are all blocked.

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

We implemented all proposed scheduling algorithms in our agent-based SWAGES environment in order to provide a (non-optimized) proof-of-concept system that tightly integrates the scheduler in the simulation environment with the parallelization and distribution algorithm. To be able to demonstrate performance gains of the proposed update strategies over the standard "cycle-based update strategy" using a practical example, we selected an actual agent-based model from a biological modeling research domain of female choice [5]. We first discuss the details of our implementation and then report the results from the empirical evaluations.

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

SWAGES is a JAVA-based agent-based simulation and experimentation server intended for any kind of computing environment (e.g., from homogeneous Beowulf clusters to heterogeneous computers connected only via the Internet). It consists of several distributed components that cooperate closely to achieve high resource utilization in a heterogeneous dynamically changing computing environment. SWAGES was used and extended to support the scheduling and monitoring of the execution of simulations for both cycle-based and non-cycle-based update strategies for agent based simulation experiments.

Without modification, SWAGES provides the communication infrastructure to start, run, and supervise simulations. It also gathers and stores simulation results in an easily accessible manner for future 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 simulation instance can run on its own host and maintains a socket connection instance for all communication purposes (e.g., information about the current simulation cycle, simulation parameters, etc. will be delivered on this connection).

SWAGES required several modifications to be able to implement and work with non-cycle-update strategies. SWAGES was extended to support the merging of distributed configuration and distributing simulation environments containing only subsets of agents across a series of processors. Associated communication support protocols were also added to provide a mechanism for simulation instances running on different processors to access the new features. In order to facilitate non-cycle-based update strategies, a single simulation instance must distribute the agents in addition to the context in which they are to run (e.g., environmental specification, agent initial conditions, agent models). This distribution could be performed inside the centralized server if the server had explicit knowledge of how to initialize an agent.

Since SWAGES can only start simulation instances, but cannot initialize agents within a simulation instance (or perform any other operation *within* simulation instances), parallelizing and distributing new simulation instances is therefore a three-step process. First, SWAGES informs a simulation instance that a resource for distributed parallel simulation is available (i.e., that there is a host computer where a new simulation instance can be run). If the simulation instance decides that it wants to split off a subset of its agents and run it on another host, it accepts the resource offer by sending back a serialized representation of all those agents that are supposed to be run in the remote simulation instance. SWAGES, in turn, launches a new simulation environment on another processor and provides it with the serialized agent set. From that point on, the local and remote simulation instances continue to update their agents, with the local instance treating the serialized agents as proxy agents and the remote instance treating all other agents as proxies. Whenever an update for a proxy agent is required by a simulation instance (because a local agent ended up in the even horizon of the proxy agent), the simulation instance will request an update from the simulation instance running the proxy agent via SWAGES.

Updated state information for any agent needs to be shared among all simulation instances. This can be done in peer-to-peer fashion using some broadcast or shared memory mechanism, or it could be accomplished using a server to broker the communication. SWAGES uses the latter mechanism and acts a global repository for the updated agent states computed in the simulation so that proxy representation can be updated on demand. The updating of a simulation instance and request for proxy agent state is accomplished in a non-blocking manner to allow agents in other instances of the algorithm to be chosen and updated during the slow I/O operation of communicating the updates. During the communication phase newly generated agent states are shared and received. The information of new update states of remote agents is stored in their respective proxy representation to be used by the simulation instance. If a requested agent state does not exist in the central repository, SWAGES will store the request until the data becomes available and also inform the simulation instance that "owns" the agent that another simulation instance requires updated state information. This information can be used to influence the update policy in **AltSched**$_G$ (e.g., in the *Remote Block* policy).

To implement the above policies and select a set of (unblocked) agents AS that can be updated, we start with a set that contains a given agent A chosen by the policy and then recursively add into A all agents within the event horizon of any agent already in the set. This final set $TC(A)$ forms a *transitive closure* of A that contains all agents that are connected via their overlapping interaction ranges and is thus update independent from all other agents in the simulation instance (i.e., agents in the set $C - AS$).[12] If AS contains a proxy agent, then AS can be updated only once until that proxy agent's state updates thus blocking all agents in AS by placing their updates on hold. Optionally, the list of agent states that were updated is sent back to the SWAGES server for bookkeeping. If no agent states are updated, a list of proxy agents causing blocking in that simulation instance is requested. The SWAGES server responds with any updated agent states requested or previously requested by that simulation instance. Additionally, a list of identifiers of agents in that simulation instance that are causing blocking in other simulation instances is also sent, which can be used in update policies.

4.2 Evaluation

We evaluated the three most promising update strategies defined above: *Remote Event First with Remote Blocks*, *Youngest First*, and *Youngest First with Remote Blocks*.[13] In addition, we included the "Cycle-Based Update Strategy" as a standard control condition. For the simulation model, we picked a realistic model from one of our current agent-based modeling domains, that of *female mate choice in treefrogs*. In this model, male and female treefrogs are located in a swamp area. Male frogs are stationary within the environment and indicate their presence and readiness for mating by repeatedly making "mating calls" of fixed, but different quality. Females initially enter the swamp from the rim, listening to the males' calls and repeatedly making choices about which of the males to approach based on the "quality" of male's mating call. Once a female has picked a male, she will approach the male based on the directions she obtains from locating the source of the male's call ("phonotaxis"). It is known from the biological literature that females show phonotaxis toward calls of males with higher *pulse numbers* (e.g., [9, 10]). Our specific model of female choice is intended to study the influence of male and female spatial distributions on mating success of females (measured in terms of the overall male fitness) when females pursue one of two "choice strategies": a *best-of-n*

[12] Another way to view this concept is to consider the collection of agents as a graph of all agents at a given time. Let each node represent an agent and directed edges represent that agent's ability to sense or influence the connecting node. All nodes that are reachable from a given node define a transitive closure. Therefore, the collection of agents in a transitive closure is a set of agents that can be updated independently from other agents in the simulation. Independent updating is possible because agents outside of the transitive closure have no ability to influence or be influenced by those agents inside the transitive closure.

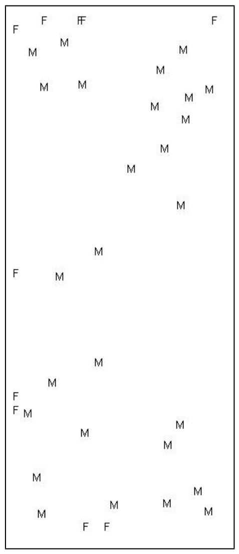
[13] *Remote Event First* was left out because it did not show sufficient performance gains in our evaluation scenario even though it might work well in others.

strategy where they pick the male with the best quality of the closest *n* males [3], or a *minthresh* strategy where they pick the closest male whose quality is above a minimum threshold [4] (for more details on the model, see [5, 2]).

For the evaluation experiments, ten females were initially positioned based on a Gaussian function along the rim of the swamp. 27 males were initially positioned by an inverse Gaussian function in the middle of the swamp (positioning more males towards the critical rim areas, see the left part of Figure 4). The swamp was modeled by a continuous rectangular area of 10×25 meters. In comparison, frogs are 4.75cm in size and can leap up to 1.44cm in one hop. While the male update function does not change the male frogs behavior unless a female frog is within mating range, in which case the male will mate, the female update function will map perceived call qualities onto the direction towards the chosen male and cause the female to leap (at a fixed speed of 1.44cm/sec) in that direction. When a female is within mating range of a male (4cm), she will attempt to mate (regardless of whether the male was the chosen one or not, which models the biologically hypothesized behavior). A simulation run starts with placing all agents (male and female) in their initial locations and updating them until all females have mated (which is always guaranteed to happen because there are more males than females in the environment).

For the evaluation of the four update strategies, we ran the same initial configuration (keeping male and female distribution fixed as well as the distribution of the male calls) under 10 different random split conditions distributed on a fixed pool of processors (i.e., 2, 4, and 8).[14] The right graph in Figure 4 shows the results, which were obtained by averaging over the total simulation run-time (from starting the SWAGES gridserver with the experiment startup file until the server quit) across the different split conditions. As can be seen from the results, there is already an immediate benefit of using any of the update strategies other than the standard cycle-based strategy with more than one processor, even though the performance gain really becomes more pronounced as the number of processors increases. In particular, in the case of 8 processors, changing from the standard update strategy reduces the overall execution by more than half for all cases. Note that the excellent performance of the Remote Event First with Remote Blocks strategy has to do with the fact the agent update sequence of this policy closely aligns with the optimal agent update sequence of this simulation. The optimal agent update sequence of a simulation is based on the characteristics of the agents in the simulation as well as the simulation's terminating condition. Since males do not and females select males based on closest proximity then the projected remote event of a male-female intersection would actually accurately measures the simulations terminating condition and therefore select the agent updates necessary to achieve the simulation termination quickly. Also, the small increase in overall run-time has to do with the additional bookkeeping required for more processors, which is due to the low run-times of *Remote Event First with Remote Blocks* that shows up explicitly while being absorbed within the run-time of the other strategies (given that it is only a small fraction).

[14] We did not include the one processor case since there is no significant performance difference between any of the employed strategies.



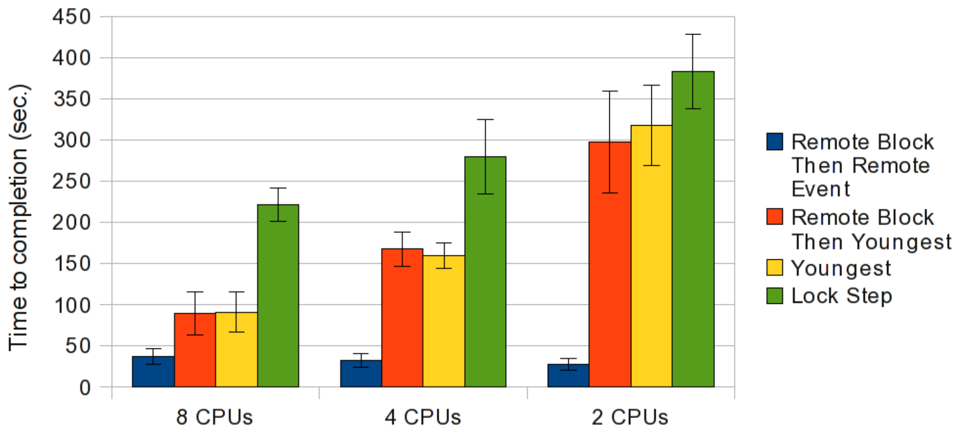


Fig. 4 The initial configuration of male and female frogs in the swamp (on the left) and the results (on the right) showing the mean overall simulation run-time (in seconds) together with their standard deviation averaged over all split conditions (see text for details).

4.3 Discussion

The empirical evaluation confirms what we were expecting based on the rationale for defining our update strategies, namely that coordinating the updates of agents in distributed simulations using simple heuristics that re-order update sequences without changing the "semantics of the simulation" (i.e., the simulation outcome) can lead to significant performance improvements in the context of parallel distributed simulations. It is important, however, to keep in mind that the exact gains of using different update strategies will depend on several factors, at least on (1) the complexity of the update functions of individual agents, (2) the agent distribution, (3) the agent interaction range, (4) the agent translation function, (5) the size of the world, etc. For the above strategies, we expect to see in general the best performance gains for more complex agents where most of the processor time is spent on agent update functions relative to the simulation book-keeping. In those cases, any way that can re-order the update sequence to give priority to agents that either require information from other simulation instances or could provide information to other simulation instances (that are required at some future point) will reduce the overall simulation run-time relative to that of the "cycle-based update strategy", which is oblivious to any interactions between parallel simulation instances (recall the example from the non-optimality proof where the "cycle-based update strategy" can lead to a lock-step behavior).

It is also important to note that while the above evaluation shows significant performance improvements over the cycle-based update strategy, there is still room for further improvement. For example, it is possible to use a finer-grained distinction about dependencies among agents that does not amount to computing the full transitive closure (e.g., an agent really only needs information for agents that can potentially interact with it at cycle n and not the full transitive closure). Hence, as long as all agents in its interaction range are at cycle n, it can be updated and by keeping its previous state at cycle n around it will allow other agents in its interaction range to update at a later point without causing inconsistencies. This type of optimization can lead to fewer blocked scenarios since there will be fewer agents identified as dependent that would therefore be required to update at cycle n before any of the agents advance to cycle $n+1$.

Another interesting question is how update strategies that are intended to reduce remote blocks can be combined with heuristics that prioritize agents based on their estimated distance to a terminal condition. It is currently unclear whether there is a general answer to this question (e.g., to always prefer advancing agents close to terminal conditions if there are guarantees for the heuristic such as being *admissible*).

Finally, we would also like to point out that richer agent-based simulations that include, in addition to agents, environmental states (e.g., global or local temperature) or other entities (e.g., non-movable, but consumable food sources) will require additional mechanisms for distributing and keeping track of those state across simulations instances effectively. This is also true of locations that can have properties assigned (e.g., swamp land vs. mountain side).

5 Conclusions

In this chapter, we investigated the utility of using novel update strategies for agents in simulations of agent-based models. These strategies differ from the standard cycle-based update strategy with respect to the update sequence of agent updates from initial to terminating conditions, but without changing any simulation outcomes. We demonstrated that the performance of parallel distributed agent-based simulations extended from our previous parallelization and distribution algorithms can be significantly improved if the proposed heuristics are employed. Specifically, we were able to achieve more than 50% shorter overall simulation run times in an agent-based simulation model taken from a biological research domain that investigates female choice behavior in tree frogs.

While any particular performance improvements will always critically depend on the nature of the employed agents, the proposed heuristics seem promising across the board. This is because they attempt to anticipate information exchanges between distributed simulation instances that will likely be required at some future time and prioritize agent updates of those agents whose state will be required.

Future work will investigate how the heuristics can be adaptively combined to utilize their individual strengths. We will also investigate ways to improve the detection of update independent subsets of agents that do not solely rely on event horizons (which are only a rough estimate of possible interactions). In particular, we are interested in exploring reflection methods that will be able to gain and utilize information in the agent update function about whether an agent is likely to interact with another agent. Finally, we will also look at replacements for the transitive closure computation which is expensive and not needed in its entirety to determine subsets of agents that can be updated.

References

1. Barwise, J., Moss, L.: Vicious Circles. CSLI Lecture Notes (1996)
2. Boyd, S., Scheutz, M., Hercog, L.: Exploring female mate choice strategies in tree frogs with a spatial agent-based model (in preparation)
3. Janetos, A.C.: Strategies of female mate choice - a theoretical-analysis. Behavioral Ecology and Sociobiology **7**(2), 107–112 (1980)
4. Jennions, M.D., Petrie, M.: Variation in mate choice and mating preferences: A review of causes and consequences. Biological Reviews of the Cambridge Philosophical Society **72**(2), 283–327 (1997)
5. Scheutz, M.: Model-Based Approaches To Learning: Using Systems Models And Simulations To Improve Understanding And Problem Solving In Complex Domains, *Modeling and Simulations for Learning and Instruction*, vol. 4, chap. Artificial Life Simulations–Discovering Agent-Based Models. Sense Publisher, Rotterdam (2008)
6. Scheutz, M., Madey, G., Boyd, S.: 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)
7. Scheutz, M., Schermerhorn, P.: Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models. Journal of Parallel and Distributed Computing **66**(8), 1037–1051 (2006)
8. Scheutz, M., Schermerhorn, P., Connaughton, R., Dingler, A.: Swages - an extendable distributed experimentation system for large-scale agent-based alife simulations. In: Proceedings of Artificial Life X, p. forthcoming (2006)
9. Schwartz, J.J., Buchanan, B.W., Gerhardt, H.C.: Female mate choice in the gray treefrog (hyla versicolor) in three experimental environments. Behavioral Ecology and Sociobiology **49**(6), 443–455 (2001)

10. Schwartz, J.J., Huth, K., Hutchin, T.: How long do females really listen? assessment time for female mate choice in the grey treefrog, hyla versicolor. Animal Behaviour **68**, 533–540 (2004)
11. Steinman, J.S.: Discrete-event simulation and the event horizon. In: PADS '94: Proceedings of the eighth workshop on Parallel and distributed simulation, pp. 39–49. ACM Press, New York, NY, USA (1994)