Adaptive parallelization of multi-agent simulations with localized dynamics

Alexandru-Ionuţ Băbeanu, Tatiana Filatova, Jan H. Kwakkel, and Neil Yorke-Smith

Delft University of Technology, The Netherlands.
{A.I.Babeanu,T.Filatova,J.H.Kwakkel,N.Yorke-Smith}@tudelft.nl

Abstract. Agent-based modelling constitutes a versatile approach to representing and simulating complex systems. Studying large-scale systems is challenging because of the computational time required for the simulation runs: scaling is at least linear in system size (number of agents). Given the inherently modular nature of MABSs, parallel computing is a natural approach to overcoming this challenge. However, because of the shared information and communication between agents, parellelization is not simple. We present a protocol for shared-memory, parallel execution of MABSs. This approach is useful for models that can be formulated in terms of sequential computations, and that involve updates that are localized, in the sense of involving small numbers of agents. The protocol has a bottom-up and asynchronous nature, allowing it to deal with heterogeneous computation in an adaptive, yet graceful manner. We illustrate the potential performance gains on exemplar cultural dynamics and disease spreading MABSs.

Keywords: agent-based models · large-scale systems · parallel computing · scientific simulation · asynchronous computation

1 Introduction

Whereas multi-agent-based simulation constitutes a versatile approach to representing and simulating social, biological and other types of (complex) systems, the use of MABS to study large-scale systems is challenging due to the computational time required. Scaling a MABS is linear or worse in the number of agents. A natural approach to handling this issue is parallel computing. Whereas parallelizing a set of independent runs of a simulation is straightforward, using multiple processing cores to accelerate a *single* simulation run is much more challenging. Despite the modular nature of MABS, agents (and other simulation components) engage in frequent interactions, so their computations cannot be parallelized in simple ways [19].

To tackle this challenge, we present a novel protocol for shared-memory, parallel execution of MABSs. We exploit the fact that a MABS may be expressed as a sequential chain of computational steps, and that MABS models often involve updates that are *localized*, in the sense of involving small numbers of agents.

Compared to most work on parallelizing MABS, our protocol handles the computation in a profoundly asynchronous manner: different CPU cores may work on different steps of the simulation at any given instant, and may handle different agents at different times. This enables adaptive adjustments to fluctuations due to heterogeneity inherent to the simulated process or the underlying infrastructure. The protocol also facilitates parallelization of MABS models that are fully sequential in form. Potential performance gains, which we illustrate using two MABSs (Axelrod-like cultural dynamics and network-based disease spreading), are a consequence of computing resources being fully used at every instant, at the cost of carrying out additional computation. We provide insights about the associated protocol overhead and guidelines for managing it.

2 Related Work

The challenge of scaling up MABS has seen recurring efforts in the literature [4, 19]. One line of work exploits the notion of a *super-agents*: subsets of agents with some similarity are combined into representative agents [21]. Along these lines, Arneth et al. [2] introduce a concept of *agent functional types*, which can be seen as super-agents corresponding to different strategies or styles. Related to but distinct from (manually-coded) super-agents, Tregubov and Blythe [24] discuss methods to automatically simplify a MABS, such as by automatic abstraction. Super-agents have been found to be problematic in spatial settings [19].

A second paradigm employs surrogate models that emulate interesting properties of MABS [10, 17]. This attracts more attention as capabilities of machine learning techniques increase [1, 18], but do not directly address accelerating a given MABS. Our ambition is to avoid any re-programming or approximation.

A third approach, which is closest to our work, aims to use high-performance computing (HPC) techniques and hardware for running MABS. Along these lines, Blandin et al. [6] present a distributed-memory, message-passing approach to parallelizing certain types of agent-based simulations in Python, illustrated with a migration model where it can handle up to 10 billion agents – while still employing certain model simplifications and abstractions designed to reduce communication between processes. Blythe and Tregubov [8] is another example of an architecture for massively distributed MABS execution, leveraging contemporary multi-processing and load balancing infrastructure, combined with graph-partitioning techniques. By contrast, Axtell [5] runs 120 million agents on a single, multi-core, shared memory system, reporting that "Many HPC architectures turns out to not be useful". Generic computational frameworks have also been developed for agent-based-like simulations in biology, such as those described in Breitwieser et al. [9] and Plesser et al. [20], which allow for jointly using distributed-memory and shared-memory parallelization.

While scale-up already demonstrated by HPC-driven efforts is remarkable, we note that most research to date is subject to a fundamental limitation: parallelization goes hand in hand with strictly splitting the computation into time steps and updating (a step-dependent subset of) all agents at each step. Dif-

ferent subsets of updates within that step are then allocated to available CPU cores, using some workload-balancing strategy. While some performance may be gained from context-dependent improvements of the allocation strategy, computing cores/nodes that eventually run out of work may not proceed to the next step until the current step has been completed. This is at odds with the local nature of many MABS models, allowing agents to update many steps ahead of other agents, as long as the former have the required input at every step. Moreover, the per-step splitting approach cannot be applied to models that lack the many-updates-per-step formulation in the first place – for instance, Axelrod-type models of cultural dynamics [3] and Markov-Chain simulations, which have a sequential, one-update-per-step formulation – but could still benefit from parallelization. We propose a generic way of jointly handling these issues, which at heart is a more flexible approach to handling *time* in scientific computing, which we present as a protocol for shared memory, adaptive parallelization.

Our proposed protocol is reminiscent of existing work on (software-based) *dynamic scheduling*, going back at least as far as Blumofe et al. [7], with a good overview of relevant tools and concepts presented by Ham [16]. The openMP library [12], which is widely used for parallel HPC in a shared memory context, also incorporates the option for dynamic scheduling. However, dynamic scheduling focuses on parallelizing computational tasks that are independent of each other, whereas our protocol is designed to gracefully handle precisely such dependence relations between tasks. Also, our protocol incorporates a clean procedure for interfacing with any given MABS model.

Sensitivity to dependence relations is present in a different approach to adaptive, asynchronous parallelization of MABS, proposed by Scheutz and Harris [22], Scheutz and Schermerhorn [23]. Their work is complementary to ours in multiple ways: 1. it uses a distributed memory paradigm with message passing between processes, while our protocol heavily relies on shared memory; 2. it ensures consistency via *proxy entities* and *event horizons*, while we ensure it via worker *records* and task *recipes*; 3. each of their CPUs handles a preallocated subset of agents at all times, while our cores may handle different agents at different times 4. it comes with a sophisticated formalism (involving constructs like bodies, sensors and actuators) in terms of which the model needs to be expressed, while our protocol is less involved and 'invasive' in that sense; 5. it is focused on spatial models with moving agents and maximum movement velocities, while our applications do not yet involve moving agents. Ideas from the two approaches could be gradually combined in future work.

3 Methodology: Computational Protocol

The central idea is that a MABS may be conceptualized as a *chain*, where each element in the chain is a *task*. Each task is a block of consecutive operations that are carried out at some point during the simulation. An operation may consist, for instance, of generating a random number or incrementing a variable. Normally, these tasks would be evaluated sequentially, based on their order in

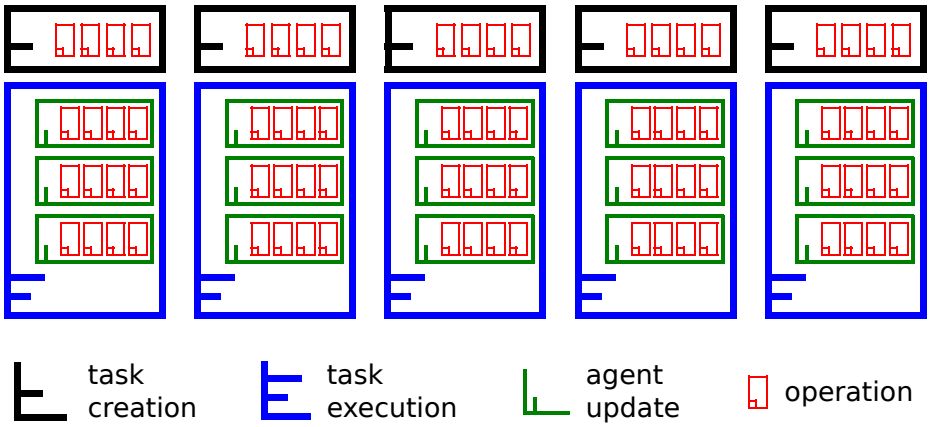

Fig. 1. Partitioning the simulation. The bottom of the figure is a legend, conveying the meaning of each type of object above, via colour and (partial) shape. Each vertical block corresponds to one task, split between a creation (black) and an execution (blue) part. The small (red) boxes denote the basic operations that make up the simulation. Within each task execution, subsets of operations make up agent updates (green).

the chain. In many cases, however, one may alter the evaluation order without changing the outcome of the simulation: there are many pairs of tasks that are entirely independent of each other, and may thus be evaluated in any order, or in parallel. For example, two agents whose next decisions are influenced only by their neighbours, and who have no neighbours in common, are independent for their next step.

Our protocol is designed to take advantage of precisely this aspect, allowing for a significant amount of parallelization, while preserving the evolution of the system. This is achieved in an adaptive manner, with multiple computational *workers* autonomously iterating the chain and executing those tasks that can be executed without violating model-dependent constraints (manifested as dependence relations between tasks). While the amount of parallelization and resulting speed-up is ultimately constrained by the number of available CPU cores (i.e., one for each worker–thread pair), there will be stronger bounds determined by the MABS *model*, the size of the system and certain choices pertaining to how the protocol is applied.

3.1 Task-based representation

Fig. 1 gives an intuitive overview of how the protocol effectively partitions the set of operations that make up a simulation. Each resulting subset of operations is associated to one task, with important constraints in place: for any two distinct operations θ and ρ associated to task α, any operation ϕ that depends on θ and that ρ depends on must also be associated to α. A simulation is mathematically equivalent to the sequence of operations obtained by concatenating all tasks,

with an ordering of updates that is consistent with all dependence relations between them. In the more specific context of a MABS, at the level of each task, many operations are typically grouped into *updates*, with each update capturing a change in the state of one or several agents.

Every task is first *created* and added to the end of the chain. Later, it is *executed* (the entailed operations are carried out) and finally erased from the chain. Creating the task already carries out part of the associated computation, while executing the task carries out the remaining part, based on information produced during creation and stored with the task. The split between creation and execution is also constrained: if operation ρ is included in the creation part, any operation θ on which ρ depends must also be included, assuming that θ is associated to the same task to start with. Still, there may be significant freedom for deciding how much computation is carried out during creation vs execution.

3.2 Dependence relations between tasks

We assume there is a finite set of variables V that can be repeatedly altered during the simulation. The execution part of each task α depends on an *input subset* of variables $I(\alpha)$ and modifies another, *output subset* of variables $O(\alpha)$; the two α-subsets may have an arbitrary amount of overlap. If α precedes another task β in the chain (at any separation) and $O(\alpha) \cap I(\beta) \neq \emptyset$, there is a *dependence* (or causal link) $\alpha \prec \beta$ between the two: the execution of an arbitrary β can only start after having completed the execution of all α with $\alpha \prec \beta$. On the other hand, if two tasks α and β have no direct or indirect dependence relation, then α and β commute: they can be executed in any order, as well as in parallel (to an arbitrary extent).

To be precise, $I(\alpha)$ and $O(\alpha)$ above are defined in a maximally conservative way: they are the subsets of variables that may conceivably influence and, respectively, be influenced by the computation encoded by α. In practice, the subsets of variables actually used/modified will typically be smaller, due to decisions made during the execution of α (this does not lead to redundant causal links: not changing something that could have been changed is also information). Note that, while conceptually useful for the explanations above, our protocol does not explicitly compute any input or output subsets when inferring dependence relations. Instead, it uses only necessary information (like agent ids) previously recorded onto the chain and workers in a dynamic fashion.

3.3 Worker-chain dynamics

Adaptive parallelization is achieved using multiple workers that operate on the chain in a simultaneous but asynchronous manner. Each worker is assigned to a dedicated thread. Workers have the freedom to execute tasks as soon as possible, provided that dependence relations between tasks are not violated, that race conditions are absent (since everything takes place within one program run with a fully-shared memory pool), and that the (unavoidable) use of mutex locks does not lead to deadlock situations. This requires a self-consistent set of rules

specifying the joint dynamics of workers and chain: a *workflow*. We focus here on a relatively simple formulation, which we use for the experiments in Sec. 4.

The chain is implemented as a bidirectional, linked list of tasks, with each task holding pointers to the previous and next tasks, so that erasing tasks inside the chain is cheap and clean. At any point in time, every worker is either waiting to enter the chain, or is located at a given task in the chain, with different workers attached to different tasks. Every worker iterates the chain from start to end. Upon reaching task β, worker w is able to infer whether β depends on any task α that has already been encountered. At this point:

- if β depends on a previous task, or if the task is already being executed by another worker u, the worker w refrains from executing β; instead, it collects the information describing β and integrates it with already collected information from previously-encountered tasks, which it can use to infer dependence relations for subsequent tasks; then w attempts to move to the next task
- otherwise, w proceeds to execute β, erase β from the chain and return to the start of the chain (while discarding all information about previously encountered task)

At most one task is created at any instant and added to the end of the chain, by a worker attempting to leave the last element. It does so by invoking a global, model-specific routine.

It is crucial that worker w cannot move to a task γ where another worker v is already located, unless the latter is already executing γ. The implied waiting of one worker behind the other is handled by a dedicated mutex lock attached to each task in the chain.

Two further mutex locks, namely an *enter-lock* and an *erase-lock* are attached to the chain itself. Specifically, the enter-lock treats the case of a new task being created when the chain is empty (with no task and no associated locks present). This situation is guaranteed to occur at the start of the process and is likely to occur many times afterwards. Hence, the new task is created by the first worker entering the chain, while holding the enter-lock. Second, the erase-lock ensures that at most one task is being erased at any given point in time (by the worker that just executed it), thus avoiding inconsistencies related to simultaneously erasing consecutive tasks.

We use the term *cycle* to refer to one round of chain exploration by a given worker, between two consecutive returns to the start of the chain. A cycle ends either when the worker has just erased a task, or when it has reached the end of the chain and cannot create a new task. The latter may happen because 1) all desired tasks have already been generated for the given simulation; or 2) the worker has reached the maximum number of tasks that it is allowed to generate per cycle. The latter tasks-per-cycle limit avoids (unlikely) situations when a worker at the end of the chain indefinitely creates new tasks that cannot be executed because of being dependent on previously-encountered tasks, thus extending the chain to unreasonable lengths and potentially increasing the exploration time for all other workers. This upper bound is controlled by a parameter.

3.4 Choices in applying the protocol

We briefly note three conceptual choices being made when applying the protocol to a given MABS model. 1. *Chain granularity* refers to the amount of computation allocated per task. 2. *Task depth* defines the split between task creation and execution. 3. *Workflow parameters* are, notably, n, the number of workers, and C, the maximum number of created tasks per cycle.

3.5 Model-specific implementation

The protocol involves a significant amount of functionality that is independent of the MABS model to which it is applied. Around this model-independent functionality we created a dedicated software framework (in due course available open source) allowing one to 'plug in' any MABS of interest. To do this, one implements a model-specific interface, according to a set of conventions that allow for the intended communication between the model and the workflow. The framework itself is implemented in C++.

The interface can be understood in terms of two generic concepts: 1. *recipe*: model-side counterpart of the task; 2. *record*: model-side counterpart of the worker. These concepts capture model-specific data-structures and functionality. The recipe specifies the kind of information the task holds after its creation, and how that is to be used for its execution. The record specifies the kind of information the worker needs to hold, in order to recognize whether the task at hand is dependent on any task previously encountered, along with the procedure for carrying out this assessment. It also specifies how the information in the task at hand is integrated with that already held by the worker (if needed) and how the worker-held information is initialized and reset at the start of a new cycle.

4 Experimental Evaluation

At this first stage of our research, the purpose of experimentation is to establish the extent to which our protocol exhibits, for a given MABS simulation, the expected speed increase as more workers are added. To achieve this, we measure, for different values of $n \in \{1 \dots 5\}$, the total simulation time T as a function of a task size proxy s. For small s little speed-up is expected. We keep $C = 6$ fixed, since separate experimentation showed its effect to be negligible. For each combination of s and n, T is averaged over 5 simulation instances with different starting seeds (and initial states, whose generation does not contribute to T).

Two experiments are reported, for two associated MABS models formulated according to different paradigms: a model with sequential interactions involving random pairs of agents in Sec. 4.1; a model where all agents are updated at each step, conditionally on nearest-neighbours' states during the previous step, in Sec. 4.2. The former is mostly known for illustrating a patter-formation mechanism, while the latter is also known for its real-world, predictive potential. The two experiments also differ in terms of details related to formulating s and using the protocol, as explained further below.

4.1 Cultural dynamics

First we apply our protocol to an Axelrod-type model of cultural dynamics [3], following the specifications in Bădeanu et al. [11]. The N agents are all connected to each other, and each is equipped with a sequence of traits, each expressed with respect to one of the F cultural features. At each step, two agents are chosen at random to interact, with preassigned roles: a *source* and a *target*. The target may then change one trait, based on a probabilistic procedure that depends on all traits of both agents, which is intended to mimic social influence.

On the protocol side, the granularity is so that each task captures one pairwise interaction. The depth is so that creation handles the random selection of the interacting pair, leaving the bulk of the interaction to the execution. Hence, the recipe holds the two agents' identifiers, while the record specifies that a task at hand is considered dependent if either the source or the target agent was a target in any task previously encountered by the worker.

Fig. 2 shows the results of the experiment, where s is defined as F, since the bulk of one interaction is built around an iteration over all features. We see that simulation time T increases with task size s, independently of n, which is in agreement with the fact that the total number of created tasks is fixed. Moreover, we see a clear trend of T decreasing with n for a fixed s, which confirms the anticipated performance increase as more workers are added. While the magnitude of this effect (in terms of the ratios between the T values of different n curves) increases with s, we also observe a saturation effect: for $s < 150$, it does not help to have more than $n = 4$ workers, while for much lower s it does not help (and it might actually harm) to have more than $n = 3$ workers. All this is consistent with the intuition that, as task size increases, the protocol overhead becomes less important. This experiment involved 2×10^6 steps per run, with the values of the remaining model parameters fixed as: $N = 10^4$, $q = 3$ (number of possible traits per feature), $\omega = 0.95$ (bounded-confidence threshold).

4.2 Disease spreading

Second we apply the protocol to a simplified SIR-type epidemic model, which captures essential aspects of predictive MABSs of contagious disease spreading, such as those used in Gemmetto et al. [14], Gomez et al. [15] for real-world scenarios. Specifically, N agents reside at the nodes of a fixed graph with constant degree k and a ring-like structure. Each agent may be in one of three possible states: susceptible (S), infected (I) or recovered (R). At each step, for each agent, a transition may occur between two consecutive states, according to simple probabilistic rules controlled by parameters $p_{\mathrm{SI}}, p_{\mathrm{IR}}, p_{\mathrm{RS}} \in (0, 1)$. The transition from S to I also depends on the fraction of infected connections.

Using the protocol involves additional subtleties compared to the first experiment. First, there are two types of tasks in the chain: the first type pertains to agents computing their new states; the second type pertains to agents replacing their current states with the new states. Second, each task handles a subset of

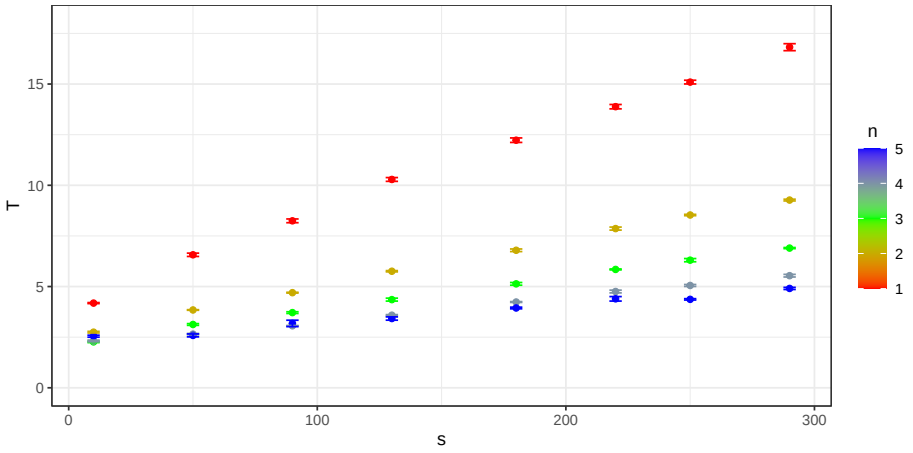

Fig. 2. Running cultural dynamics with our protocol. Showing the average simulation time (vertical axis) as a function of task size proxy (number of features F, along horizontal axis), for different numbers of workers (legend). Vertical error bars show the standard mean error range based on five simulation instances.

agents, using a partition of the system into equal subsets, which is fixed throughout the simulation. The aggregation level inherent to this partition specifies the chain's granularity. In terms of depth, each creation produces a recipe containing the agent subset identifier, along with a binary flag indicating the task's type, leaving the actual computations for the execution. The record specifies that: a second-type task should not be executed before a previously-encountered first-type task handling the same agent subset; a first-type task should not be executed before a previously-encountered second-type task handling the same or a connected agent subset. Connections between agent subsets are encoded in an aggregate graph computed once (just after generating the initial state); this computation contributes to the measured T (unlike generating the initial state).

Fig. 3 shows the results of the experiment. Unlike Sec. 4.1, there is no model parameter that can serve as s. Instead, we define s as the size of the agent subsets, which directly controls the task size, as well as the chain's granularity. A consequence of this, immediately visible in the figure, is that T does not increase with s (as it does in Fig. 2), exhibiting instead a sharp decrease, followed by a stabilization: the model-related computation is the same for all s, just differently partitioned, so the fine-grained partitioning inherent to $s < 50$ becomes very taxing due to protocol overhead, regardless of n. In the stabilization region we can see the anticipated performance increase with increasing n, with saturation reached for $n = 4$. As we go deeper into the low-s region, the saturation is reached for lower n, and we clearly see that adding workers may decrease performance. This experiment involved 3×10^3 steps per run, with the values of the remaining model parameters fixed as: $N = 4 \times 10^3$, $p_{\mathrm{SI}} = 0.8$, $p_{\mathrm{IR}} = 0.1$, $p_{\mathrm{RS}} = 0.3$, $k = 14$.

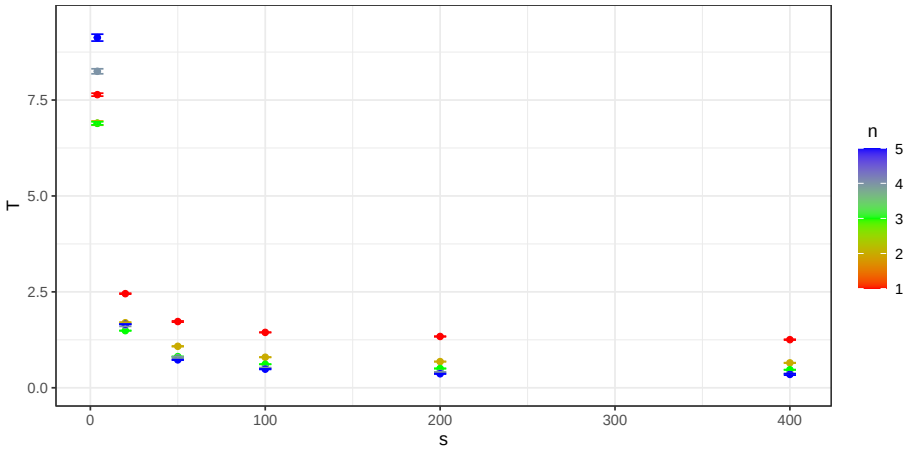

Fig. 3. Running disease spreading with our protocol. Showing the average simulation time (vertical axis) as a function of task size proxy (number of agent updates per task, along horizontal axis), for different numbers of workers (legend). Vertical error bars show the standard mean error range based on five simulation instances.

5 Summary and Future Work

To unlock the potential of multiple cores for accelerating single runs of a MABS, we presented a novel protocol for shared-memory, parallel execution of multi-agent-based simulations. The benefits are seen in experiments on Axelrod-like cultural dynamics and network-based disease spreading simulations.

While our protocol can in principle be applied to any form of computation, it makes sense to use it only if that computation is CPU-bound (rather than input–output-bound), as can be the case for MABS with many agents in, for instance, ecologoy [13]. Moreover, we expect an actual speed-up of the computation if, for some choice of chain granularity and task depth, the resulting (directed, acyclic) graph of dependence relations between tasks is sparse enough. In other words, a typical task α should only depend on a small part and only affect a small part of the system. In the context of MABS, 'small' typically means that the relevant fractions of input and output variables $I(\alpha)/V$ and $O(\alpha)/V$ scale as $1/N$, where N is the number of agents. This is what we mean by *localized* dynamics – the term *local* is better suited for more specific MABS where information propagation is constrained by an underlying geometry. For instance, models involving agents on a lattice that only interact with nearest-neighbours are good candidates for our protocol. Even if the model involves global entities coupled to the entire system, the protocol could still be beneficial if the associated interactions are relatively rare, and/or the coupling is only one-way (only influencing or being influenced by the rest of the system).

Future work will present applications of our protocol to simulations with non-stationary agents or dynamic spatial geometry. Second, we consider more

explicitly using the DAG nature of the computation, which could reduce the overhead of the protocol in terms of both memory and CPU usage. Third, we plan to invoke techniques from Scheutz and Harris [22] and join the strengths of the approaches. Beyond MABS, we expect that other types of (non-agent-based) simulations could also benefit from this line of research, including some types of Markov-Chain simulations.

Acknowledgements This work benefited from interactions with J. Decouchant and M. Mircea, and was partially supported by the TU Delft Institute for Computational Science and Engineering, and by TAILOR, a project funded by the EU Horizon 2020 programme grant 952215. We also thank two anonymous referees for their thoughtful comments on an early version of this article.

References

1. Angione, C., Silverman, E., Yaneske, E.: Using machine learning as a surrogate model for agent-based simulations. PLOS ONE **17** (2022), https://doi.org/10.1371/journal.pone.0263150
2. Arneth, A., Brown, C., Rounsevell, M.: Global models of human decision-making for land-based mitigation and adaptation assessment. Nature Climate Change **4**, 550–557 (2014), https://doi.org/10.1038/nclimate2250
3. Axelrod, R.: The dissemination of culture: A model with local convergence and global polarization. Journal of Conflict Resolution **41**(2), 203–226 (1997), https://doi.org/10.1177/0022002797041002001
4. Axtell, R., Farmer, J.: Agent-based modeling in economics and finance: Past, present, and future. Tech. Rep. 2022-10, Institute for New Economic Thinking at the Oxford Martin School (2022)
5. Axtell, R.L.: 120 million agents self-organize into 6 million firms: A model of the U.S. private sector. In: Proc. of AAMAS'16, pp. 806–816 (2016)
6. Blandin, N., Colglazier, C., O'Hare, J., Brenner, P.: Parallel python for agent-based modeling at a global scale. In: Proc. of CSSSA'17 (2017), https://doi.org/10.1145/3145574.3145588
7. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Proc. of PPOPP'95, pp. 207–216 (1995), https://doi.org/10.1145/209936.209958
8. Blythe, J., Tregubov, A.: Farm: Architecture for distributed agent-based social simulations. In: Proc. of Second International Workshop on Massively Multiagent Systems, pp. 96–107 (2018)
9. Breitwieser, L., Hesam, A., de Montigny, J., Vavourakis, V., Iosif, A., Jennings, J., Kaiser, M., Manca, M., Meglio, A.D., Al-Ars, Z., Rademakers, F., Mutlu, O., Bauer, R.: BioDynaMo: a modular platform for high-performance agent-based simulation. Bioinformatics **38**, 453–460 (2021), https://doi.org/10.1093/bioinformatics/btab649
10. ten Broeke, G., van Voorn, G., Ligtenberg, A., Molenaar, J.: The use of surrogate models to analyse agent-based models. J. Artif. Soc. Soc. Simul. **24**(2) (2021), https://doi.org/10.18564/jasss.4530

11. Băbeanu, A.I., van de Vis, J., Garlaschelli, D.: Ultrametricity increases the predictability of cultural dynamics. New Journal of Physics **20**(10), 103026 (2018), https://doi.org/10.1088/1367-2630/aae566
12. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE **5**(1), 46–55 (1998)
13. Filatova, T., Voinov, A.A., van der Veen, A.: Land market mechanisms for preservation of space for coastal ecosystems: An agent-based analysis. Environ. Model. Softw. **26**(2), 179–190 (2011), https://doi.org/10.1016/j.envsoft.2010.08.001
14. Gemmetto, V., Barrat, A., Cattuto, C.: Mitigation of infectious disease at school: targeted class closure vs school closure. BMC Infectious Diseases **14**(1), 695 (2014), https://doi.org/10.1186/s12879-014-0695-9
15. Gomez, J., Prieto, J., Leon, E., Rodríguez, A.: Infekta – an agent-based model for transmission of infectious diseases: The COVID-19 case in Bogotá, Colombia. PLOS ONE **16**(2), 1–16 (2021), https://doi.org/10.1371/journal.pone.0245787
16. Ham, D.R.: Dynamic scheduling in multicore processors. Ph.D. thesis, The University of Manchester (United Kingdom) (2012)
17. Lamperti, F., Roventini, A., Sani, A.: Agent-based model calibration using machine learning surrogates. Journal of Economic Dynamics and Control **90**, 366–389 (2018), https://doi.org/10.1016/j.jedc.2018.03.011
18. Leeuw, B.D., Ziabari, S.S.M., Sharpanskykh, A.: Surrogate modeling of agent-based airport terminal operations. In: Proc. of MABS'22, pp. 82–94 (2022), https://doi.org/10.1007/978-3-031-22947-3_7
19. Parry, H.R.: Agent-based modeling, large-scale simulations. In: Complex Social and Behavioral Systems: Game Theory and Agent-Based Models, pp. 913–926, Springer (2020), https://doi.org/10.1007/978-1-0716-0368-0_9
20. Plesser, H.E., Eppler, J.M., Morrison, A., Diesmann, M., Gewaltig, M.O.: Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In: Proc. of Euro-Par'07, pp. 672–681 (2007)
21. Scheffer, M., Baveco, J., DeAngelis, D., Rose, K., van Nes, E.: Super-individuals: a simple solution for modelling large populations on an individual basis. Ecological Modelling **80**(2–3), 161–170 (1995)
22. Scheutz, M., Harris, J.: Adaptive scheduling algorithms for the dynamic distribution and parallel execution of spatial agent-based models. In: Parallel and Distributed Computational Intelligence, pp. 207–233, Springer (2010), https://doi.org/10.1007/978-3-642-10675-0_10
23. Scheutz, M., Schermerhorn, P.W.: Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models. J. Parallel Distributed Comput. **66**(8), 1037–1051 (2006), https://doi.org/10.1016/j.jpdc.2005.09.004
24. Tregubov, A., Blythe, J.: Optimization of large-scale agent-based simulations through automated abstraction and simplification. In: Proc. of MABS'20, pp. 81–93 (2020), https://doi.org/10.1007/978-3-030-66888-4_7