International Workshop on Massively Multi-Agent Systems (MMAS2018)
in conjunction with IJCAI/ECAI/AAMAS/ICML 2018

FARM: Architecture for Distributed Agent-based Social Simulations

Jim Blythe and Alexey Tregubov

Information Sciences Institute, Marina del Rey CA 90292, USA
blythe@isi.edu, tregubov@usc.edu

Abstract. In many domains, high-resolution agent-based simulations require experiments with a large number (tens or hundreds of millions) of computationally complex agents. Such large-scale experiments are usually run for efficiency on high-performance computers or clusters, and therefore agent-based simulation frameworks must support parallel distributed computations. The development of experiments with a large number of interconnected agents and a shared environment running in parallel on multiple compute nodes is especially challenging because it introduces the overhead of cross-process communications.
In this paper we discuss the parallel distributed architecture of the FARM agent-based simulation framework for social network simulations. To address the issue of shared environment synchronization we used a hybrid approach that distributes the simulation environment across compute nodes and keeps the shared portions of the environment synchronized via centralized memory storage. To minimize cross-process communication overhead, we allocate agents to processes via a graph partitioning algorithm that minimizes edge cuts in the communication graph, estimated in our domain by empirical data of past agent activities. The implementation of the toolkit used off the shelf components to support centralized storage and messaging/notification services.
This architecture was used in a large-scale Github simulation with up to ten million agents. In experiments in this domain, the graph partitioning algorithm cut overall runtime by 67% on average.

Keywords: Agent-based modeling and simulation · Parallel distributed simulation · Large-scale simulation · Parallel distributed computing

1 Introduction

The development of large-scale agent-based simulations for social science can require significant computational resources. In order to simulate social networks such as Facebook, Twitter or Github with high resolution, including models of individual users, one may need to simulate hundreds of millions of agents. Each agent may have a complex behaviour model that requires a significant amount of information from the environment. A large number of complex agents exchanging information with the shared environment inevitably consumes a large amount of computational power.

The efficient simulation of large models often requires more computational resources than are available to the researcher on one machine. Therefore, it is often necessary to utilize resources of high performance super computers or clusters. Clusters require parallel computations. There are two main technical reasons to use clusters of computers and parallel computations: (1) to make simulations run faster and (2) to overcome the memory limitations of a single machine. Running an agent-based simulation on such platforms requires support of parallel computations in the architecture of the simulation framework. In this paper, we present FARM, a distributed parallel agent simulation architecture with centralized storage and messaging services. FARM's distributed architecture was designed to addresses parallel simulation challenges such as a synchronized shared environment and cross-process communication overhead. As part of its solution, FARM uses a combination of centralized and distributed storage. Centralized storage is used for portions of the environment that must be shared by agents on different compute nodes, and the local memory of a node is used in all other cases. Since communication between compute nodes (cross-process communication) uses the network, it significantly slows the overall simulation time. We discuss methods, including graph-based agent partitioning and smart lazy communication, to minimize cross-process communication to reduce simulation runtime and increase the size of simulations that can be run in practice.

FARM has been used to run simulations approaching 10 million agents on 10 nodes with 16GB RAM, or on 4 nodes with 64GB RAM, simulating 30 million events in approximately one hour of real time. The novel contributions of FARM include (1) explicit reasoning about centralized and distributed storage at run time and (2) graph-based partitioning of agents between compute nodes based on empirical communication data. FARM's distributed parallel architecture was utilized to overcome memory limitation of compute nodes at expense of cross-process communication overhead. The impact of cross-process communication overhead was reduced by optimized graph-based partitioning of agents between compute nodes.

We present empirical results from a simulation of millions of Github user agents on the performance of the partitioning algorithm. In our experiments, the algorithm cuts overall runtime by around 67% on average and cuts the amount of cross-process communication required by 70%. We conclude with a discussion of next steps to further improve performance at very large scale, enabling high-fidelity agent simulations to be applied to city-scale or nation-scale domains.

2 Related work

Two types of parallel processing are described in the agent-based simulation literature: (1) simulations that run a large a number of experiment trials in parallel and (2) simulations that run one large computationally complex experiment trial in parallel on several compute nodes. Many distributed parallel agent-based simulations were originally developed using the first approach [3, 4, 10]. However, in such domains as transportation, where a large number of active agents is neces-

sary in each experiment run, the resources of one computer are often not enough even for one trial of the experiment. For example, to simulate air traffic over the US models need to handle more than 40000 flights [12] In such simulations, agents usually follow some simple rules to model physical interactions of the real world, and experiments often need hundreds of thousands of them.

The development of parallel distributed simulations, exploiting the power of parallel computation, has been used to approach scalability of multi-agent simulations [4]. In the distributed simulation models described in [5, 11, 12] agents are partitioned by geographical location where partitions allowed concurrent execution. When agents move due to their actions from one region to another they are reallocated to a process corresponding to a new region or partition of the environment. For example, Šišlák et al. [11] proposed an architecture for distributed parallel simulation of air traffic control using the locality of interactions among agents and the environment to distribute agents across several computers. General purpose agent-based simulation frameworks also heavily rely on the locality of agent interactions with their environments (e.g. the agent only partially observes the environment). For instance, in Repast the grid-locality of communications with the environment is a part of the framework for parallel simulations [4]. This principle of distributing agents across compute nodes according to their location and sphere of influence is common among agent-based simulations in social, economic and climate studies [6, 10].

Compared with transportation or economic domains, simulations of social networks with high resolution may require an even larger number of agents. Social networks such as Facebook and Twitter have hundreds of millions of monthly active users. Additionally, communication among social network users is often not bounded by geographical location, and so partitioning agents and the environment in distributed parallel simulations also requires a different approach. In Github simulation experiments, discussed in the following sections, we applied graph-based partitioning of agents between compute nodes based on empirical communication data.

3 FARM distributed simulation architecture

FARM supports distributing agent simulations across multiple compute hosts in parallel to enable large-scale multi-agent simulations. It supports two cases of distributed processing for multi-agent simulations: one case in which a single simulation is distributed across multiple compute nodes, and one in which an experiment consists of many repeated trials, each a separate simulation, which are distributed across multiple compute nodes. Each compute node is an independent process running on its own host. FARM includes support for explicitly representing experiments consisting of multiple trials, iterating over the set of independent variables, in support of a hypothesis. However, in this paper we focus on support for a single simulation distributed across multiple hosts, since the trials within an experiment are independent of each other, and hence simpler to allocate from a computational standpoint.

FARM is implemented in Python and designed to support simulations involving DASH agents. DASH is a platform for developing cognitive agents, also written in Python [2]. During a simulation, DASH agents can either affect each other directly, through peer-to-peer communication, or indirectly through one altering some part of the shared state that the other perceives. This indirect communication is moderated through a DASH communication hub, that receives information from agents about actions that were taken and passes back the observable results of the action, while modeling the world state within the simulation. However, FARM can be used with simulations involving other kinds of agent representations.

We assume that each compute host will run many agents within the simulation in a single image, with inter-agent communication between agents on the same compute host supported efficiently by shared memory within the image. FARM utilizes centralized storage and messaging service to synchronize the environment that agents share in simulation. FARM provides tools for automated management of simulation parameters and allows the researcher to control experiment setup.

Figure 1 shows the distributed architecture of FARM and how its components are allocated across multiple compute nodes. Each compute node runs a process, called DASH Worker, that runs a subset of agents and maintains a synchronized view of the part of the simulation environment that must be shared between nodes.

On each DASH Worker, the part of the state that is used by only agents on that DASH Worker is kept locally, while the rest is shared, and maintained via an in-memory database. Information about the state updates is distributed via messaging services. FARM uses Apache ZooKeeper [1] to provide a fast in-memory database for small transactions and messaging/notification services for task distribution and synchronization.

FARM uses the DASH agent-based simulation framework, where DASH agents communicate with each other and interact with their shared environment via communication hubs. DASH communication hubs accept action specifications from agents and return their observable effects, while maintaining the shared state of the environment as a result of these actions. Each compute node has at least one communication hub. Communication hubs synchronize their portion of the environment with other hubs in the network as needed via Apache ZooKeeper.

Typically the environment represents some shared resources of the simulation (e.g. posts, pages, pictures on social networks, source code repositories on Github, etc.). FARM distributes agents and environment resources (via communication hubs) across multiple compute nodes. Communication between agents and shared resources of the environment can therefore be viewed as an agent-to-resource graph. For example, in the Github experiment, discussed below, the shared environment consists of software repositories that agents observe and contribute to, and the agent-to-resource graph is then a user-to-repository communication graph (labeled as U-R graph in figures). This is a bipartite graph

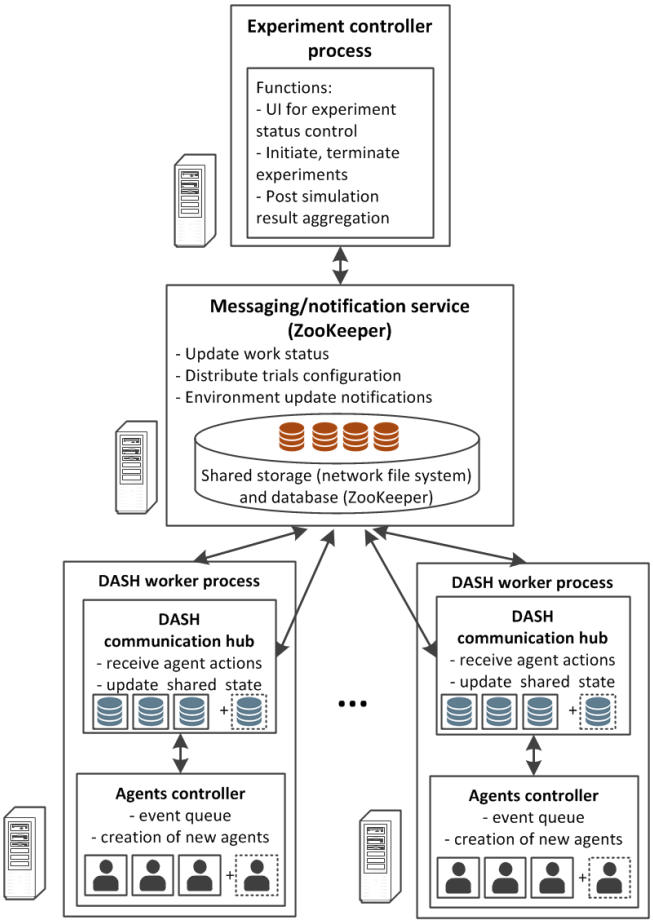

Fig. 1. The FARM distributed simulation architecture.

of agents and shared resources; two vertices in the graph are connected when agents access the shared resource. Frequencies of interaction between agents and shared resources can be used as weights in the graph.

The agent-to-resource graph is partitioned when agents are allocated to compute nodes. Cross-process communication in the simulation then corresponds to the case where two agents allocated to different compute nodes access the same resource. If information about expected agent communication is available, it can be used to partition an agent-to-resource graph in a way that reduces this cross-process communication during simulations. Past communication history between agents and the environment can be used to build an agent-to-resource graph. Graph partitioning algorithms that reduce the total flow across edge cuts can then be used to partition the agent-to-resource graph and find an efficient allocation of agents to compute nodes.

In FARM we utilized a k-way graph partitioning algorithms implemented in METIS [8,9]. METIS uses multilevel partitioning algorithms that reduce the size of the graph by collapsing vertices and edges, partition the smaller graph, and then uncoarsen it to construct a partition for the original graph. METIS utilizes novel algorithms developed by Karypis and Kumar [8,9]. These algorithms allow parallel work on large graphs as well as multi-constraint partitioning.

4 Github simulation and experiment setup

In this section we discuss a Github simulation model (shown in Figure 2), which was implemented using the distributed architecture presented above. Github [7] is a hosting platform for software repositories using the git version control software, that provides additional features such as wikis. Github is an example of a social network where users can comment on commits, make pull requests, fork repositories, create branches, etc. There are several dozens of millions of users and repositories on Github. Our Github simulation is capable of running several million agents and repositories simultaneously on multiple compute nodes.

The DASH architecture provides the necessary components to model social networks. In our Github experiment, DASH agents represent Github users, and DASH communication hubs model the social network infrastructure. Communication hubs provide access to a shared state of the environment — the Github repositories. The state of repositories reflect the history of other users' actions on a particular repository, and it can drive next actions of an agent. For example, if one user submits a pull request, another user may respond to that request by accepting or rejecting it.

The goal of the Github experiment was to simulate interactions between agents and repositories. This also allowed us to test the performance of the simulation framework in different configurations — using different agent partitioning techniques and different number of compute nodes in different trials.

We measured the overall simulation runtime and the number of cross-process communications, which is the number of times compute nodes synchronized the state of their repositories.

In this experiment an agent-to-resource graph is called user-to-repository graph. To compare the quality of an user-to-repository graph partitioning techniques, we measured the number of edge cuts in the partitioned graph. The number of graph partitions is the same as the number of compute nodes in this experiment.

We compared two partitioning algorithms: random user-to-node allocation and multilevel k-way graph partitioning using the METIS library [8, 9]. In random user-to-node allocation, the number of users per node was balanced, and all repositories that are only accessed by one user were allocated to the same node as the user. This means that only repositories that are accessed by two or more users could belong to edge cuts.

We used all user activities on Github for one month as a training dataset, modeling the actions taken such as forking a repository or committing code, but

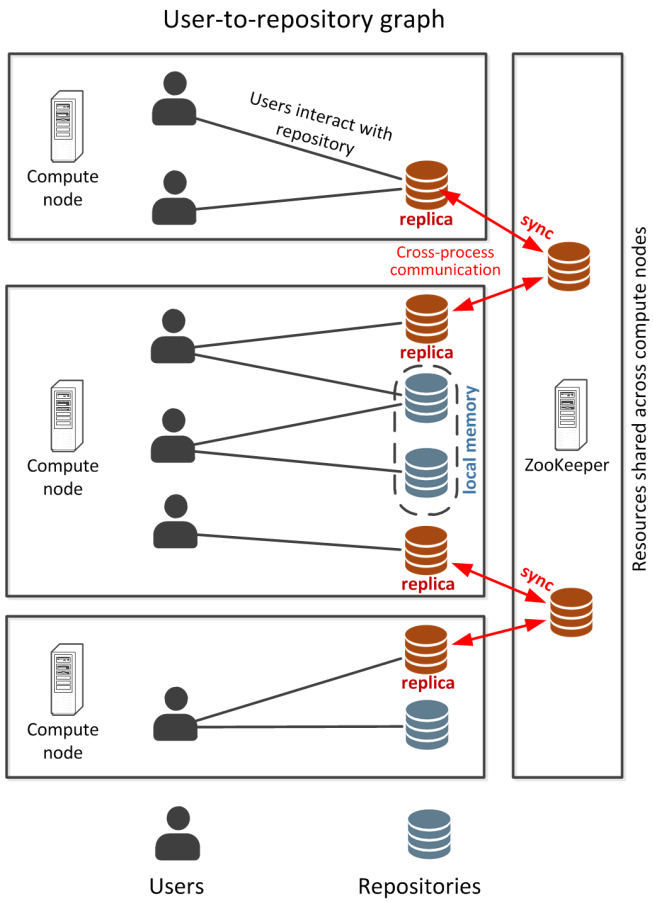

Fig. 2. The distributed Github simulation allocates each agent to a single compute node. Repositories are modeled locally if they affect only agents from one compute node, otherwise they are replicated.

not storing the content of the code that was committed. This provided the initial state of the simulation with 1.8 million users and 3.2 million repositories. We chose this size of the initial state as a realistic representation of the scale required to model active users and repositories on Github over a reasonable time period. The simulation ran on a cluster of 18 compute nodes, each with 16GB of RAM and 4 core CPUs. However, in order to measure cross-process communication overhead in multiple trials, we designed a reduced model with simplified agents so that the entire simulation could fit into 16GB of RAM on one compute node. These simplified agents did not use the cognitive modeling capabilities that are available in DASH and had a relatively restricted memory of their history of Github activity.

We also ran trials in which the state of the repositories shared between partitions was not synchronized (labeled as 'no sync') to measure results without the cross-process communication overhead. These simulations were, of course, incorrect, but they enabled us to measure task allocation and aggregation separately from the cross-process repository communication.

5 Results

Figure 3 shows simulation runtime of all 36 trials. In most of the trials, METIS user-to-repository graph partitioning ran more than three times faster than random partitioning. METIS user-to-repository graph partitioning algorithm allowed 3% load imbalance and was configured to minimize the number of edge cuts. It is important to note that minimizing edge cuts in user-to-repository graph is an approximation to minimizing the number of repositories shared between several partitions.

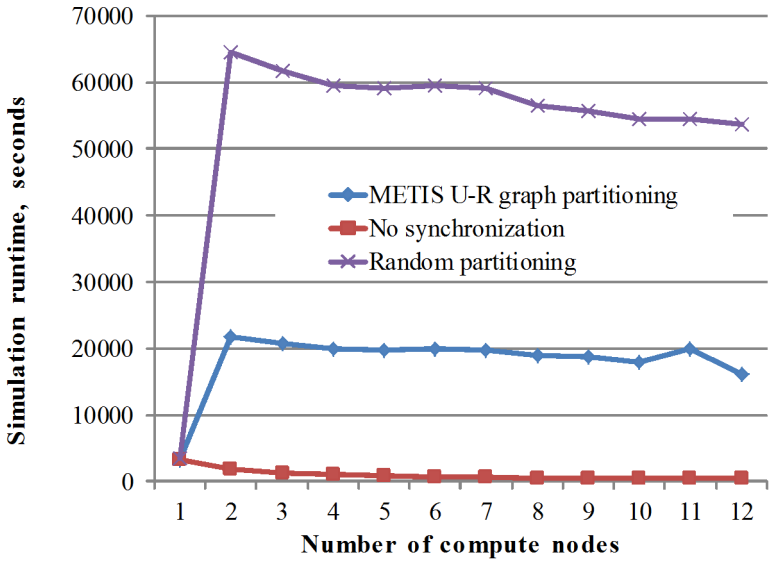

Fig. 3. Simulation runtime with different partitioning algorithms. In order to test results with fewer nodes, simplified agents have been used that can fit in 16GB of RAM on a single machine. In more faithful experiments, a cluster of 18 nodes was used.

Runtime decreases with the number of available computation nodes. However, due to cross-process communication overhead, the runtime of trials with several compute nodes is always significantly higher (10-20 times) than the runtime of a trial running one just one node. In general, it is not always feasible

to fit the whole simulation model and all its agents in one compute node. More complex agent behavior requires more computational time and memory, which may not be available on one machine.

Figure 4 shows the number of cross-process communications in trials with a varying number of compute nodes. We can see that cross-process communication with random partitioning increases only by 5% when the number of compute nodes goes from 2 to 12 nodes. For METIS k-way graph partitioning it increases by 48 %, and it remains mostly the same when more than 9 nodes used. This means that after some number of nodes the number of cross-process communications does not increase.

On average, in our domain, cross-process communications constituted 31% of user to repository interaction under a random user allocation, and 9% under user allocation that used METIS k-way graph partitioning.

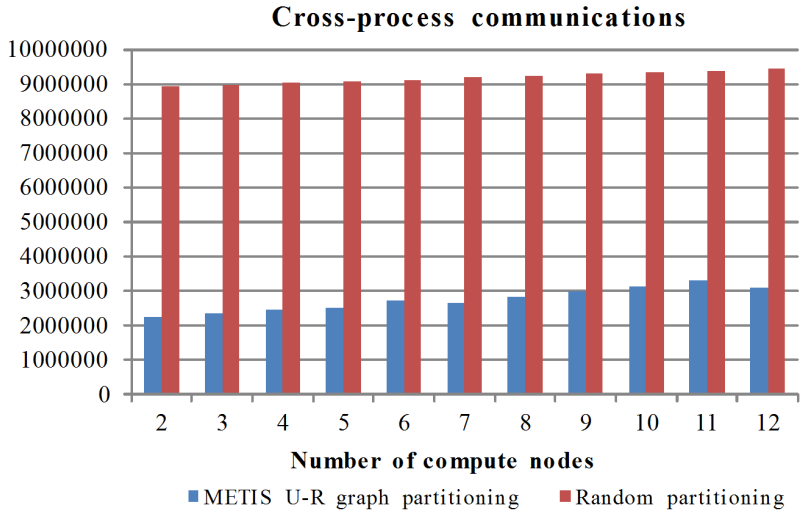

Fig. 4. Total number of cross-process communications in the simulation.

Figure 5 shows the number of edge cuts produced by partitioning. We tested user-to-repository graph partitioning while varying the number of partitions from 2 to 18. Random partitioning produces almost the same number of edge cuts regardless of the number of partitions, whereas for METIS k-way graph partitioning, the number of edge cuts levels off after 10 partitions. In all cases, the METIS k-way graph partitioning produces at least three times fewer edge cuts. This is consistent with the runtime, since the number of edge cuts directly impacts the amount of cross-process communication and runtime.

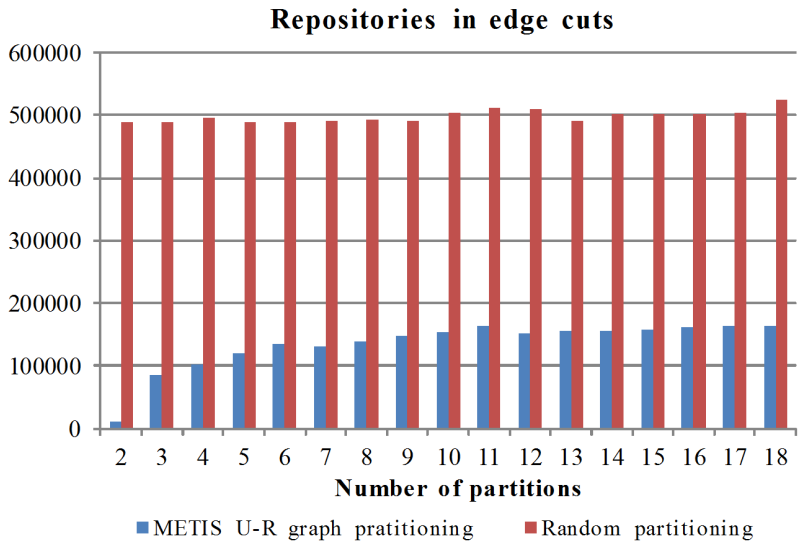

Fig. 5. Total number of repositories that belongs to more than one partition (edge cuts). METIS multilevel k-way graph partitioning and random partitioning.

6 Discussion

One of the main difficulties of distributed parallel simulations is maintaining shared state synchronization, which requires cross-process communication. The amount of cross-process communication significantly slows down the overall simulation runtime. Partitioning of the agent communication graph can significantly reduce the amount of cross-process communications and runtime.

In this research we conducted experiments using the distributed FARM architecture with a hybrid of centralized and distributed storage for shared state synchronization. We measured performance of different agent-to-node allocation strategies. In our Github simulation experiment, the METIS multilevel k-way graph partitioning reduced the number of cross-process communications and runtime more than three times compare to a balanced random agent-to-node allocation.

This distributed architecture is applicable for different domains. In the Github experiment we used the history of user-repository interactions to allocate agents to compute nodes. Different domains might use different agent partitioning techniques, for example, the geographical location of agents. If the simulation model allows identifying highly interconnected clusters of agents that mostly share only some small portion of the environment (e.g. in Github experiment it is a group of users that mostly only work their own repositories), then the hybrid storage architecture of DASH can take advantage of this to reduce cross-process communications and runtime.

Since network communication between nodes is the biggest contributing factor to simulation runtime, it might be reasonable to sacrifice some partition size balancing to reduce cross-process communication. In our experiments, the user-to-repository communication graph was partitioned by the METIS k-way partitioning algorithm with only 3% percent of partition size imbalance allowed. We plan to explore the trade-off between partition size balancing and amount of cross-process communications in the future experiments. Additionally, since agents (e.g. Github users) and shared resources of the environment (e.g Github repositories) will typically require different amount of memory, it is also worth experimenting with balancing the numbers of agents and shared resources of the environment separately.

In this work, we did not consider dynamic reallocation of agents although this is common in other domains, particularly where allocation is location-based and agents move. An equivalent reallocation might be made in simulations of social networks when an agent changes the mix of communications, e.g. message boards or repositories, over time so that more communication is now with another host than with the host where the agent resides. Rules for such reallocation are more complex than in the case of location-based agents, however, and the benefits not always as clear.

There are benefits and limitations to the centralized storage architecture that we have discussed. Compared to fully distributed peer-to-peer communication the centralized storage and messaging service are conceptually easier to use and maintain when a distributed simulation model is being developed. Additionally, there are many off-the-shelf components available to make the implementation efficient. Large-scale simulations also require a very high throughput from the database and messaging service, which can make them a bottleneck. Apache ZooKeeper allows distributed installations and also supports local caching and delayed/lazy synchronization transparent for the simulation model, which can be used to fine-tune the simulation performance.

In large-scale simulations with several million agents running on multiple nodes, it is likely to find clusters of agents that share a similar or identical internal state. In such cases, it may be possible to compress the representation of the internal state of these agents to make the simulation more memory efficient. This in turn allows more agents to be allocated to the same node, which can reduce the total number of compute nodes and cross-process communications. We are currently exploring performance optimizations such as these in FARM.

References

1. Apache: Apache zookeeper. https://zookeeper.apache.org (2018)
2. Blythe, J.: A dual-process cognitive model for testing resilient control systems. In: 5th International Symposium on Resilient Control Systems. pp. 8–12 (Aug 2012)
3. Chow, K.P., Kwok, Y.K.: On load balancing for distributed multiagent computing. ieee transactions on parallel and distributed systems **13**(8), 787–801 (2002)
4. Collier, N., North, M.: Parallel agent-based simulation with repast for high performance computing. Simulation **89**(10), 1215–1235 (2013)

5. Cosenza, B., Cordasco, G., De Chiara, R., Scarano, V.: Distributed load balancing for parallel agent-based simulations. In: Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on. pp. 62–69. IEEE (2011)
6. Deissenberg, C., Van Der Hoog, S., Dawid, H.: Eurace: A massively parallel agent-based model of the european economy. Applied Mathematics and Computation **204**(2), 541–552 (2008)
7. Github: Github software development platform. http://github.com (2018)
8. Karypis, G., Kumar, V.: Multilevel algorithms for multi-constraint graph partitioning. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing. pp. 1–13. IEEE Computer Society (1998)
9. Karypis, G., Kumar, V.: Multilevelk-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed computing **48**(1), 96–129 (1998)
10. Scheutz, M., Schermerhorn, P., Connaughton, R., Dingler, A.: Swages-an extendable distributed experimentation system for large-scale agent-based alife simulations. Proceedings of Artificial Life X pp. 412–419 (2006)
11. Šišlák, D., Volf, P., Jakob, M., Pěchouček, M.: Distributed platform for large-scale agent-based simulations. In: Agents for Games and Simulations, pp. 16–32. Springer (2009)
12. Tumer, K., Agogino, A.: Distributed agent-based air traffic flow management. In: Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems. pp. 255:1–255:8. AAMAS '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1329125.1329434, http://doi.acm.org/10.1145/1329125.1329434