Scalability in the MASON Multi-agent Simulation System

Haoliang Wang, Ermo Wei, Robert Simon, Sean Luke, Andrew Crooks, David Freelan
George Mason University, Washington, DC, USA
Email: {hwang17, ewei, simon, sean, acrooks2, dfreelan}@gmu.edu

Carmine Spagnuolo
Università degli Studi di Salerno, Salerno, Italy
Email: cspagnuolo@unisa.it

***Abstract*—This paper describes Distributed MASON, a distributed version of the MASON agent-based simulation tool. Distributed MASON is architected to take advantage of well known principles from Parallel and Discrete Event Simulation, such as the use of Logical Processes (LP) as a method for obtaining scalable and high performing simulation systems. We first explain data management and sharing between LPs and describe our approach to load balancing. We then present both a local greedy approach and a global hierarchical approach. Finally, we present the results of our implementation of Distributed MASON on an instance in the Amazon Cloud, using several standard multi-agent models. The results indicate that our design is highly scalable and achieves our expected levels of speed-up.**

I. Introduction

MASON is an open source agent-based modeling (or ABM) simulation toolkit written in Java, and has enjoyed significant popularity in the agent-based modeling community. Though it has extensions for certain areas (such as geographic information systems, social networks, or rigid body kinematics), MASON is meant to be very general and domain-inspecific: it has been used significantly in agent-based models ranging from the social sciences to swarm robotics to population biology.

MASON was designed with efficiency in mind, and works well with large models. For example MASON was recently used in a 10-million-agent model of permafrost thawing and its consequences on Canadian communities [1]. MASON is also designed to be highly flexible and capable of being used in unusual circumstances. Along these lines, we have used MASON running in real time on-board cooperative soccer-playing robots during the RoboCup soccer competition [2].

Introduced in 2003, MASON introduced many then-unique features (for the ABM community). These included multi-threaded models, total separation of model and visualization, fully self-contained models, model serialization, 3D visualization, limited guarantees of replicability, and a small, clean model core which was orthogonal and consistent. Such capabilities are prosaic by the standards of the general simulation field, but in the ABM community they were entirely novel. MASON has since had a significant impact on the ABM modeling community going forward.

978-1-5386-5048-6/18/$31.00

In 2013 the National Science Foundation sponsored a workshop bringing together MASON's developers and the user community. This resulted in nine recommendations to improve MASON and to take it in important future directions. Foremost among these was the recommendation to add parallel and distributed capabilities for performance and scalability. This paper describes our efforts to achieve this specific objective.

Our approach builds upon well-known principles of Parallel Discrete Event Simulation (PDES) for distributed multi-agent systems [3]. We aim to produce a high performing and easy to use version of MASON, running in an enterprise or cloud environment, that can be used by researchers across a broad section of the scientific, engineering, GIS, and computational social science communities. Our working name for the distributed extensions to MASON is simply Distributed MASON.

The paper proceeds as follows: Section II provides some background and discusses related work in Agent Based Modeling applications and issues in scaling up multi-agent simulation systems. Section III describes how we are designing Distributed MASON and gives an example of an implementation of a GIS within the Distributed MASON framework. Section IV details our approach to data management, followed by our approach to load balancing in Section V. Section VI shows the results of performance evaluation running on Amazon Web Services (AWS), and Section VII offers some final observations.

II. Background and Related work

A. Agent-based Modeling Toolkits

In an agent-based simulation, multiple *agents* manipulate the world in response to information received about the world. To do this, agents or agent events are repeatedly placed on a schedule, then stepped either in parallel or in randomly shuffled order. It is not required for an agent to be embodied in the world, but this is often the case: for example swarms of robot agents collaborating to build a house, or agents as migrating spice caravans eventually forming the historic silk road. These concepts are not special to ABMs: indeed video games had been doing similar things for decades prior; and robotics simulation toolkits have often employed the same general approach, as have simulations hosted on cellular automata.

Many of the traditions and conventions of agent-based modeling can be traced to the seminal system SWARM [4]. SWARM emphasized simple single-process, single-thread models, often with agents embedded in a 2D array physical environment (a "gridworld"). There have been many SWARM-like toolkits since then, the most well known being StarLogo [5], NetLogo [6], Ascape [7], and early versions of Repast [8]. These toolkits tied the model to visualization and emphasized ease of model development over efficiency.

MASON was among the first ABM toolkits aimed at more significant model development efforts, emphasizing large, complex simulations that might be run many times. MASON emphasized efficiency, multithreading, replicable results, model/visualization separation and serialization, a wide variety of visualization facilities, and ease of customization and extension. Later toolkits (such as newer versions of Repast) have since continued in this vein. However these toolkits (including MASON) have largely been confined to a single process and memory space.

As hardware has gotten cheaper, ABM models have been able to become progressively larger and more complex, giving rise to a new trend in ABM toolkits which provide high performance distributed simulation, such as FLAME [9], Repast HPC [10], and D-MASON [11]. One ABM direction of note has been the creation of models embedded in geographic information systems (or GIS). GIS models are often complex and very large, and so provide both challenges and good motivation for distributed simulation. MASON has a full-featured GIS simulation facility called GeoMASON [12].

B. Load Balancing and Data Partitioning in Distributed Simulation Systems

Our focus in this project is to speed up the execution of a MASON simulation by employing techniques from Parallel and Distributed Simulation that were first proposed in the 1970s. The basic idea is to partition the simulation model to be able to run concurrently on multiple processors. Within this context load balancing has long been recognized as a critical technique for improving the performance of distributed simulation systems [13], [14].

Other work involves developing algorithms for finely tuning process migration strategies. For instance, a load balancing scheme aimed at a High Level Archeture (HLA) uses estimates of time delay and time gain factors to more precisely re-distribute load [15]. The work in [16] describes a method to parallelize a large-scale epidemiologic ABM developed using the Repast HPC toolkit. The authors show that by using a 128 node cluster they can achieve speedups over around 1300%.

In terms of distributed MAS simulation, a variety of approaches have been proposed, such as a combination of both an agent-based and a discrete-event simulation model [17]. There has been a growing interest in the integration of GIS and MAS (e.g. [18]–[20] and now several open source MAS toolkits now offer support the integration of geographical information into their simulations (e.g. NetLogo [6], GAMA [21], Repast [8], and MASON (see [22] for a review). This allows researchers not only to link their models to actual geographical locations but also to simulate various processes of a diverse group of objects that are impacted by space and observe the resulting spatial patterns over time. Such applications range from the studying past civilizations; to the spread of diseases; to analysis of crime, riots and conflict [23]–[27].

III. Distributed MASON Architecture

The primary motivation to parallelize and distribute MASON is to provide simulation support for massively scaled simulation systems involving upwards of tens or even hundreds of millions of agents. One of the goals is to take advantage of relatively inexpensive compute clusters now commonly available to researchers through cloud services such as AWS and Azure, or at the University or Laboratory level. Here we provide a description of the MASON architecture and the challenges of transforming it to a PDES environment.

A. MASON Design

The single threaded version of MASON was developed using a Model-View-Controller (MVC) architecture. MASON is actually divided into two parts, a visualization portion and a model portion, and the MVC architecture allows complete separation between these two portions. The model portion is entirely encapsulated in a single top-level object which has no back-pointers to the visualization.

The heart of the MASON model portion is a real-valued time schedule. The schedule allows agents to register themselves to be called at some time in the future. Models also typically hold one of more *fields* to represent some form of space. Each field is a data structure that logically relates objects and values required by the modeler. MASON provides built-in field structures in the form of square or hex grids, continuous space, graphs, and multigraphs. Fields can be 2D or 3D, bounded, unbounded, or toroidal, or entirely user-defined. Finally, MASON models are fully serializable and self-contained, and can be run side-by-side in multiple threads or in the same thread.

The visualization portion typically contains a *console* which enables the user to control the simulation schedule and various global parameters, plus one or more windows (called *displays*) which allow the user to view and manipulate data field representations. To do this, displays call upon one or more *field portrayals* which can visualize, inspect, and manipulate each field representation in a wide variety of ways: these in turn call upon *simple portrayals* which do the same for individual objects or data stored in the fields.

B. From MASON to Distributed MASON

We are presently testing Distributed MASON on three models drawn from MASON's standard model archive: *HeatBugs*, *Flockers*, and *CampusWorld*. These three models are spatially organized and so are good targets for distribution, and they vary significantly from one another in important and useful aspects. We describe them here.

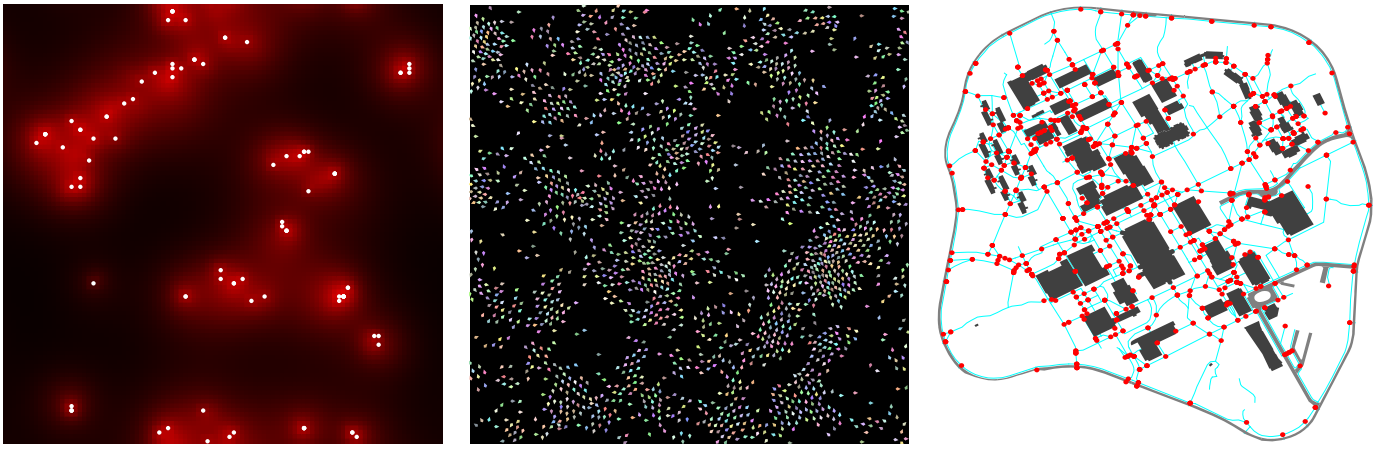

Fig. 1. MASON's HeatBugs, Flockers, and CampusWorld models.

a) HeatBugs: Introduced in the SWARM toolkit, this model is effectively the "Hello World" model of agent-based modeling. In HeatBugs, an $N \times N$ toroidal grid environment is populated by some M bugs. Each grid square can hold zero or more bugs and has a current heat value. Bugs heat up the grid square they are on, and this heat both evaporates and diffuses to neighboring squares. The bugs do not like it to be too cold or too hot, and wander to neighboring squares to hill-climb to their preferred temperature.

MASON's non-distributed HeatBugs model is implemented using a 2D array of doubles (the heat) and an overlaid 2D toroidal sparse grid of objects (the bugs). A sparse grid uses a hash table to relate objects to locations rather than storing them in a 2D array. The bugs are also agents on the schedule, and an additional agent on the schedule (the *diffuser*) is responsible for globally evaporating and diffusing heat. Each bug must be able to read the heat values of neighboring cells.

b) Flockers: This is an implementation of the classic Boids flocking model [28]. Here, some B flocking robots (the "boids") move about on a continuous 2D toroidal space $C \times C$ in size. An additional $D \ll B$ randomly distributed boids stay "dead" (immobile) and serve as obstacles for the others. Each boid maintains statistics on nearby boids (such as their locations and headings) and uses these statistics to build five directional vectors which represent behaviors such as avoidance of others, flock cohesion, and so on, then moves along the a weighted sum of the vectors.

MASON's non-distributed Flockers model is done in a 2D continuous space. MASON's continuous field places each object into a sparse grid (discretizing the continuous space) and additionally associates them with a floating-valued location. The boids are stored in this space and are also placed on the schedule. Each boid must be able to perform neighborhood lookups for nearby boids some fixed distance from him.

c) CampusWorld: This model uses MASON's GeoMASON toolkit to define an (effectively) 2D continuous space populated by buildings, roads, sidewalks, and students. Buildings and roads are arbitrary 2D shapes, the sidewalk network consists of paths in the form of sequences of straight lines (called *linestrings* in GIS parlance), and students are point objects. The students move randomly but are constrained to move along pathways only. Students do not collide with one another. MASON's CampusWorld map is a GIS representation of the George Mason University campus.

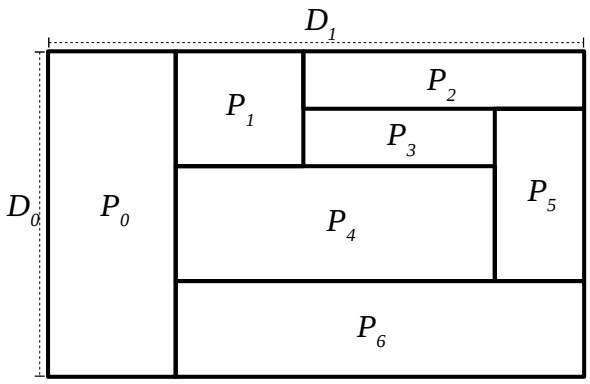

Fig. 2. Illustration of a field and its partition.

GeoMASON uses a combination of MASON data structures (such as 2D continuous space) and special geometric data structures provided by the JTS Topology Suite toolkit [29]. Unlike the Flockers and HeatBugs model, many objects in the CampusWorld model (notably sidewalks and roads) may span the entire environment. Students must be able to identify the sidewalk nearest to them so as to follow along it.

IV. Distributed Data Management

In Distributed MASON, the field is partitioned into several axis-aligned (hyper)rectangular regions, as shown in Figure 2. Following standard PDES practice we partition the network in a set of concurrently executing Logical Processes (LPs). Each LP holds one region and processes all the agents located in that region.

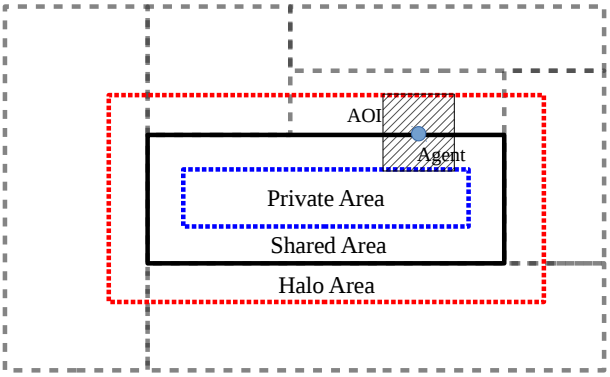

Fig. 3. Illustration of the Halo and Shared Area of a partition.

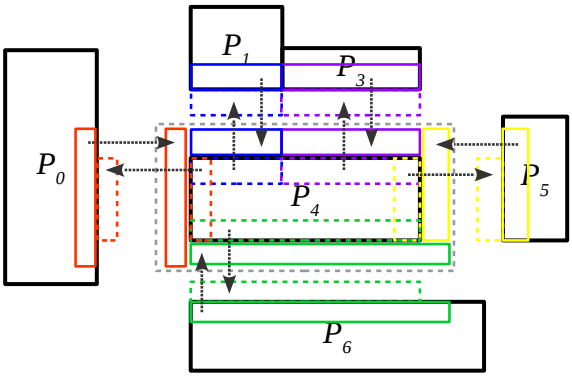

Fig. 4. Illustration of Halo Exchange.

Message exchange between LPs can be implemented either with a centralized message broker or in a peer-to-peer fashion. In our preliminary experiments with D-MASON, which uses the former approach, the centralized message broker quickly becomes a performance bottleneck as the number LPs or messages increases. Hence in Distributed MASON we use the peer-to-peer approach and try to use the communication between neighbors whenever possible for better scalability. We use the OpenMPI toolkit [30] to support many of the distributed features and algorithms in our implementation.

For most of the time, an agent is assumed to read only nearby data (defined as its *Area of Interest* (AOI)), and modify only data at its location. The system supports the rare cases when an agent needs to access and modify data in a remote location, but with a significant performance penalty.

The simulation and synchronization between LPs is done in a time-stepped fashion. For each time step, each LP will process the agents in its corresponding region that are scheduled for that time step. All the message exchanges, including agent migration, halo exchange, and remote access, will performed once the processing of the agents is complete. The synchronization between LPs is achieved at the end of each time step through OpenMPI API calls. Usually synchronization is achieved implicitly by the OpenMPI neighbor collectives in a decentralized way, but in certain cases a global synchronization using OpenMPI global collectives or barrier calls is also used.

A. Halo Exchange

To support rapid access of data stored within an AOI, each LP not only stores the data in its own region, but also maintains a cache of the data in its surrounding neighbors. The cached area is called the *Halo Area*. Part of the LP's own data will also be cached by its neighbor LPs, and that part is called the *Shared Area*. The sizes of these two areas are defined by AOI. Figure 3 provides an example of the Halo and Shared Area of a partition. Algorithm 1 shows the overall Halo Exchange algorithm.

After each simulation step, each LP will pull the data from its neighbors into its Halo Area and at the same time send the data in the shared area to its corresponding neighbors. This process is called *Halo Exchange* and is shown in Figure 4.

Our implementation supports both grids and continuous fields and both primitive value and object types. Primitive type data can be transferred between LPs directly. For objects, serializations and de-serializations are required for communication between LPs. Serialization is supported in our system with either the default Java serialization routines or with (faster) custom user-defined routines which potentially do not require building an object graph.

Algorithm 1 Halo Exchange

▷ *Each LP initializes first*

```
1:  if not initialized then
2:      myPart ← getPartition(self)
3:      myHalo ← expand(myPart, aoi)
4:      for each nid ∈ findNeighbors(self) do
5:          neighborPart ← getPartition(nid)
6:          neighborHalo ← expand(neighborPart, aoi)
7:          sendRegions[nid] ← myPart ∩ neighborHalo
8:          recvRegions[nid] ← myHalo ∩ neighborPart
9:      initialized ← true
```

▷ *Halo exchange*

```
10: sendBuf, recvBuf ← initBuffers()
11: for each nid ∈ findNeighbors(self) do
12:     sendBuf[nid] ← packRegions(sendRegions[nid])
13: MPINeighborAllToAll(sendBuf, recvBuf)
14: for each nid ∈ findNeighbors(self) do
15:     unpackRegions(recvBuf[nid], sendRegions[nid])
```

B. Remote Access

Access to data outside an agent's AOI is supported in our system via Remote Procedure Calls (RPCs) between LPs. Read access to the field data is provided in a synchronous fashion, meaning that the caller will get the value once the RPC call returns. One pitfall here is that the order of the read access from other LPs and modifications from its own LP is undefined, which may potentially cause a inconsistent view of the field.

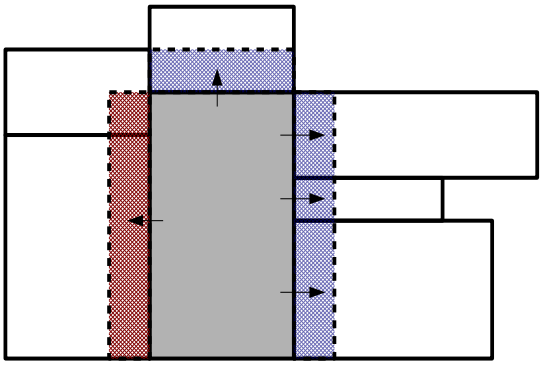

Fig. 5. Options for local load balancing.

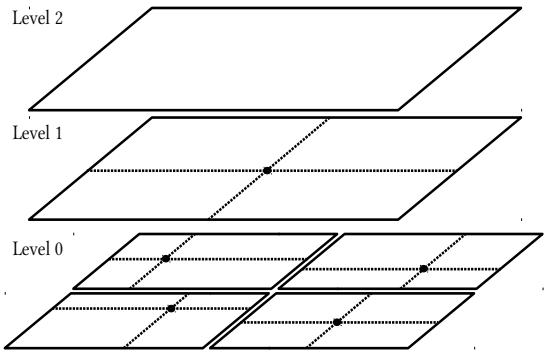

Fig. 6. Quadtree partitioning.

To avoid such an inconsistency, each LP caches the state of the field in the previous step to serve the read access from other LPs so that the modifications at the current step will not interfere with that access. On the other hand, write access to the field and access to agent states is supported in a asynchronous way. Essentially each LP has a *mailbox*. When an agent tries to modify the state of the field or another agent stored on a remote LP, such modification request will be encapsulated into a message and sent to the target LP. All the messages will get delivered by the end of the current round and will be processed at the start of next round by designated agent on each LP.

C. GIS Data

As one of major areas where MASON is being used (social and geographical simulations) often involves a large number of geometric objects in the model. In most cases, these objects are static and read-only, but may span many LPs, e.g., rivers, country boundaries, roads. In Distributed MASON, instead of partitioning static and immutable objects into different LPs, each LP will have a complete copy of all the static geometric objects in the space so that the extra communication can be eliminated at the cost of slightly more memory usage. For mutable GIS data, if the object can be abstracted into a point-object, it can be easily handled in the same way as other regular point-objects (agents, field cells, etc.) in MASON. For mutable volumetric GIS data, e.g., a polygon with some mutable attributes, such object will be stored in one master LP while other LPs covered by it only store a reference to the object. All the read and write requests to the object will be handled in the same way as the aforementioned remote access mechanism, despite that the object may be in the neighborhood.

V. Load Balancing

A balanced workload among all processors is critical to simulation performance. In this section, we will introduce the load balancing strategies used in Distributed MASON.

Our goal for load balancing is to distribute the workload among LPs such that the runtime of each step for all LPs is roughly the same, so that faster LPs do not waste time waiting for slower LPs to complete. Determining an optimal load-balancing is a well-known NP-Complete problem. In Distributed MASON we have implemented several heuristics that work at different levels and with different effectiveness and overhead. The goal at the lower levels is to balance the load frequently with minimum overhead, while the goal at higher levels is to distribute the workload as evenly as possible.

A. Local

In local load-balancing, each node tries to balance the workload *locally* with its neighbors. To do this, each node first measures its runtime for every step. When a node performs load-balancing, it collects the runtimes from its neighbors, and based on the runtimes, it chooses the neighbor and action (either expand or shrink its region) such that the variance of runtimes among the node and all its neighbors is minimized. When making load-balancing decisions, we assume the runtime is linear in the size of the region, i.e., if we shrink the region by 30%, its runtime is expected to become 30% lower as well. The overall procedure is shown in Algorithm 2.

Each partition adjustment can only shift the border by at most the AOI to avoid additional data exchange between nodes since each node already has part of its neighbors' data in its halo area. This restriction may seem to slow down the load balancing but we argue that this is actually preferable because, by limiting the adjustment and avoiding additional data exchange, the overhead is minimized and therefore the local load balancing can be done more frequently, better adapting to the change of workload.

Despite its simplicity and low overhead, there are limitations to local load-balancing. First, to avoid expensive coordination, we must constrain things such that a LP and its neighbors may not perform load-balancing at the same time. To do this, we implemented a graph coloring algorithm in the system so that at each step, only nodes with a designated color may balance their loads with neighbors. Second, since each region is (hyper)rectangular and each LP can only hold one region, the boundary shift can only be done when both the source and the destination are *aligned*. This is shown in Figure IV-C, where the blue regions correspond to viable load-balancing actions while the red region is not viable. This can result in a local optimum where loads are poorly balanced.

Algorithm 2 Local Load Balancing

```
1: runtimes ← exchangeNeighborRuntime()
2: myPart ← getPartition(self)
3: myAction ← None
4: if isMyTurn() then
                    ▷ For each dimension, among the neighbors
                      that are aligned with me, find the one
                      whose runtime differs from mine the most
5:     target ← 0
6:     maxDiff ← 0
7:     for d ← 0 to getNumDimensions() do
8:         neighbors ← findNeighborsInDimension(self, d)
9:         for each nid ∈ neighbors do
10:            if isAligned(myPart, getPartition(nid), d) then
11:                delta ← aoi[d] / getSize(myPart, d) ×
12:                  (get(runtimes, self) − get(runtimes, nid))
13:                if |delta|> maxDiff then
14:                    target ← nid
15:                    maxDiff ← |delta|
   ▷ Generate load balancing action between self and target
16:    myAction ← generateAction(target)
                    ▷ Exchange the actions with all other
                      nodes and apply all the actions
17: actions ← MPIAllGather(myAction)
18: for each action ∈ actions do
19:    apply(action)
                    ▷ Synchronize the data in the Halo area
20: HaloExchange()
```

Algorithm 3 Hierarchical Load Balancing

```
   Input: level
1: group ← getGroup(level)
2: myNode ← getNode()
                    ▷ Every node in the group sends its
                      runtime and data to the group master
3: runtimes ← collectRuntimesTo(getMaster(group))
4: data ← collectDataTo(getMaster(group))
                    ▷ The group master calculates the new centroid
5: if isGroupMaster(myNode, group) then
6:     centroid ← initEmptyCentroid()
7:     for each node ∈ getLeaves(myNode) do
8:         centroid ← centroid + center(node) ×
9:                   get(runtimes, node) / sum(runtimes)
                    ▷ Every node updates its
                      partition based on the new center
10: newCentroids ← MPIAllGather(centroid)
11: for each newCentroid ∈ newCentroids do
12:    moveCenter(newCentroid)
                    ▷ Distribute the data to all nodes
                      based on the new partition
13: distributeFrom(getMaster(group, data))
                    ▷ Synchronize
14: HaloExchange()
```

B. Global

As mentioned before, local load-balancing approach can get stuck in local optima and so we need to periodically redistribute the workload globally. Currently global load-balancing is implemented in the centralized fashion. One LP collects the runtime information and regional data from all the LPs. Various load-balancing heuristics can then be executed by that LP to calculate a new near-optimal partitioning scheme. Finally the new scheme and the corresponding data will be distributed back to all other LPs and the simulation will continue.

The large amount of data transfer in this process may seem to incur a high overhead, but this can work well in practice for two reasons. First, most simulations using MASON come with a GUI and associated visualizations, which already require constant collection of data to one node. Second, many effective load-balancing algorithms, if implemented in a distributed way, may require significant coordinations and many small message exchanges among all the nodes, which is far less efficient than the few bulk transmissions in our approach.

C. Hierarchical

The local and global load balancing algorithms discussed so far are at two ends of the overhead vs. effectiveness spectrum. In practice, it is preferable to have a tunable algorithm capable of balancing the two. So far we have assumed the entire field can be arbitrarily partitioned into axis-aligned (hyper)rectangular regions. We can constrain the unlimited partitioning flexibility by making use of a tree structure. By using data structures such as such as K-D trees or quadtrees, load balancing can be done quite effectively in a hierarchical fashion.

Consider quadtree-based partitioning as shown in Figure IV-C. Every node in a quadtree is responsible for a rectangular region. Nonleaf nodes have four children, and partition their region into four subregions, each assigned to one child. For load balancing, each region is only in charge of adjusting the partition between its four subregions.

We have implemented a quadtree-based scheme by assuming that the tree will always be full. Therefore we only need to balance the load within the a given ply, instead of between two plies. Since one region corresponds to one LP in our system, the assumption of a full tree imposes restrictions on the number of LPs that can be used by our system. For example, in case of a quadtree, the number of LPs must be a power of four. To alleviate this issue, we allow nodes at the second last level to either have four leaves (like an ordinary quadtree) or two leaves (like a K-D tree) so that the system can fully use powers of 2 LPs.

Like the local load balancing algorithm, the hierarchical algorithm performs load balancing based on the runtime of the LP. The node will first collect the runtime of its children. Then it will calculate the centroid of the children's centers using

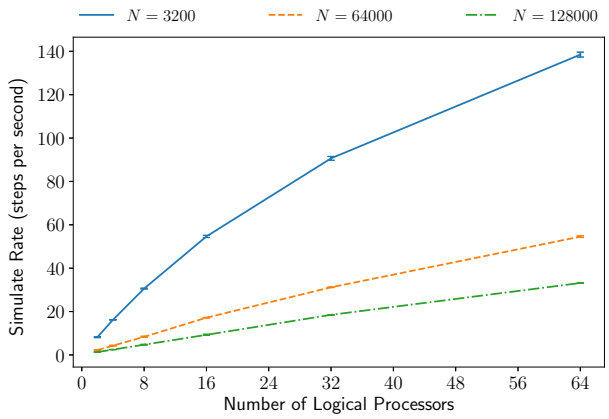

Fig. 7. Simulation rate with different numbers of processors and number of agents for the DHeatBugs model.

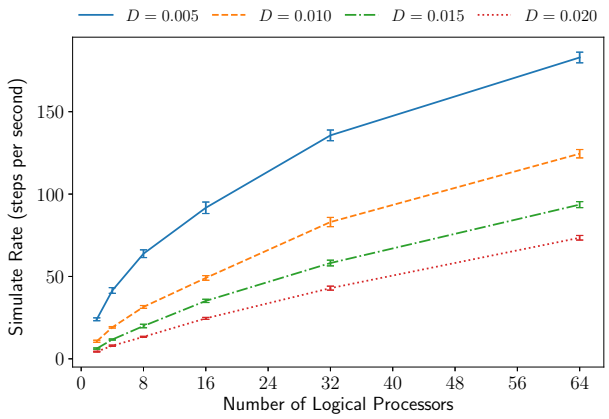

Fig. 9. Simulation rate with various number of processors and agent densities for the DFlockers model.

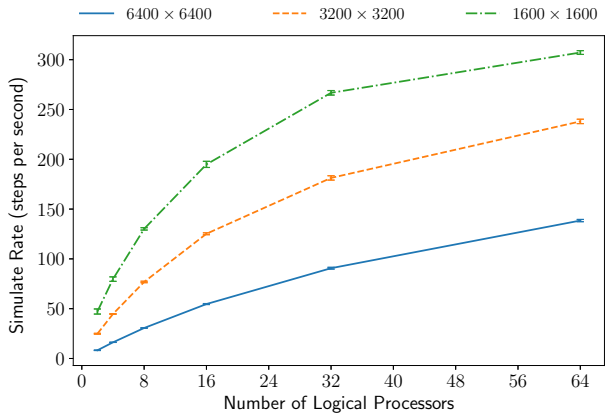

Fig. 8. Simulation rate with different number of processors and field sizes for the DHeatBugs model.

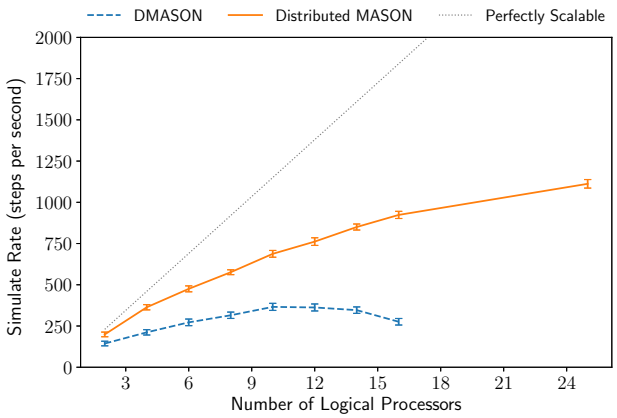

Fig. 10. Simulation rate with various number of processors for the DFlockers model in D-MASON and Distributed MASON.

their runtimes as weights. The result will be used as the new origin for the node's partitions. The runtime of this node will be set to the maximum of its children's runtimes. To apply the changes, the node will first collect all the data from its children and redistribute the data based on the new partition scheme.

Balancing higher-level (that is, more global) plies in the tree will be much more expensive, so they are performed at a correspondingly lower frequency than lower levels. The exact frequencies of load balancing at different levels is determined dynamically by comparing the estimated performance gain after balancing and the overhead it may incur. The entire procedure, shown in Algorithm 3, describes the process of hierarchical load balancing.

VI. Evaluation

In this section, we evaluate the performance of Distributed MASON using Amazon Web Services. Since our system uses time-stepped synchronization and therefore is more tightly coupled, here we focus evaluation on a more parallelized scenario where all the EC2 instances used are the same type and in the same network. The EC2 instance used in our experiments is the *c5.large* type (2 vCPUs and 4GB RAM) running 64-bit Ubuntu 16.04 LTS. OpenJDK 1.8.0 and GCC 5.4.0 are used to compile the system, OpenMPI 3.1.0, and its Java bindings. Each EC2 instance holds only one MPI slot and corresponds to one logical processor. All the results are obtained through 10 repeated experiments with different random seeds and 95% confidence intervals are provided in all applicable figures, even though some of them are too small to be visible. The metric of interest is the average *simulation rate*, defined as the number of steps executed per second. Unless specified otherwise, the field in partitioned uniformly and we calculate the average simulation rate using the time consumed by running the simulation for 2000 steps.

The test problems are distributed versions of the three previously described demo applications implemented in Distributed MASON, called *DHeatBugs*, *DFlockers*, and *DCampusWorld* respectively. We first provide the scalability result for these problems under different workload configurations, and then show the effectiveness of two load balancer implementations: *Greedy* and *Hierarchical*, when running in *DHeatBugs*.

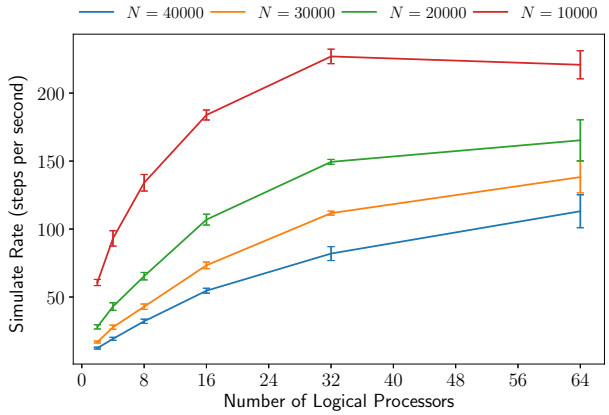

Fig. 11. Simulation rate with various number of processors for the DCampusWorld model.

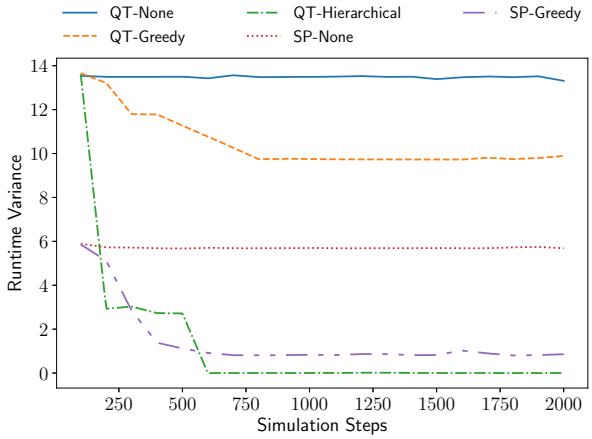

Fig. 12. Variance among nodes' runtimes as the simulation progresses with various load balancers and initial partition schemes, in the DHeatBugs model.

A. Scalability

We first evaluate the scalability of the *DHeatBugs* application implemented in Distributed MASON under different numbers of agents and field sizes, and the results are shown in Figures 7 and 8, respectively. In both Figures the X-axis represents the number of logical processors used for the simulation and the Y-axis represents the resulting simulation rate.

In Figure 7, we fix the field size to 6400×6400 and vary the number of agents from 3200 to 128000. We can see from the Figure that the simulation rate increases nearly linearly with the number of processors in all three settings. In Figure 8, we fix the number of agents to be 3200 while using various field sizes from 1600×1600 to 6400×6400. Under relatively heavy workload (6400×6400 field size), the simulation rate scales almost linearly. As the workload intensity becomes lighter, the synchronization and network communication overhead introduced by the distributed facilities becomes more and more dominant, counterweighting the performance advantage brought by more processors. Therefore the simulation rate increases sub-linearly in cases of 3200×3200 and 1600×1600 field sizes.

We now evaluate the scalability of the *DFlockers* application. In this experiment, we choose a fixed field size (1000×1000) and various numbers of agents (5000 to 20000) to obtain the result for various agent densities D from 0.005 to 0.02 agents per unit square. The result is shown in Figure 9. Similar to the result of *DHeatBugs* and as would be expected, Distributed MASON exhibits sublinear scalability when the workload intensity is quite low (e.g., $D = 0.005$ and $D = 0.01$). As the workload intensity increases, the simulation rate of *DFlockers* scales more and more linearly as the the number of logical processors increases.

Since the *DFlockers* application is also implemented in D-MASON, we compare the simulation rate for the DFlockers application in both D-MASON and Distributed MASON. In this experiment, we use a 10000×10000 field and 10000 agents. The *Perfectly Scalable* line is obtained by multiplying the simulation rate of the single-node Flockers application in MASON by the number of the logical processors used. As shown in Figure 10, the MPI-based Distributed MASON outperforms D-MASON significantly. As the number of processors increases, the centralized message broker used in D-MASON becomes a bottleneck while the MPI-based Distributed MASON uses point-to-point communications and provides better simulation rate. In fact, with our specific configuration, the ActiveMQ message broker used by D-MASON fails to process all the messages in time when there are 20 or more processors. The simulation then becomes extremely slow and often crashes. Hence only the result with 2–16 processors are shown for D-MASON in Figure 10.

Finally, we evaluate the scalability of Distributed MASON in a more complex application: *DCampusWorld*. We set the field size to be 3000×3000 and the number of agents to be from 10000 to 40000. The result is shown in Figure 11. Similar to the results obtained for *DHeatBugs* and *DFlockers*, the scalability of the system improves as the intensity of the workload increases. For the lightest workload ($N = 10000$), the distributed performance gain becomes saturated with 32 processors and, with 64 processors, the synchronization overhead overcomes the performance advantages, yielding a slight lower simulation rate even with more processors. However for the highest workload ($N = 40000$), the performance scales well with the number of processors, where with 32 processors we can get an approximately 13 times the acceleration of the single-node result.

B. Load Balancing

Here we evaluate two load balancers: *greedy* and *hierarchical*. As discussed before, the greedy balancer running on each node attempts to shift the partition boundaries with its local neighbors with the goal being to minimize the runtime variance between neighbors. The hierarchical balancer works only on top of a quad-tree partitioned field. Each node in the tree structure will be in charge of balancing the runtime of its four children through moving the center that divides its region. The balancer can be triggered when the estimated performance gain after the balancing exceeds its overhead. In our experiments, however,

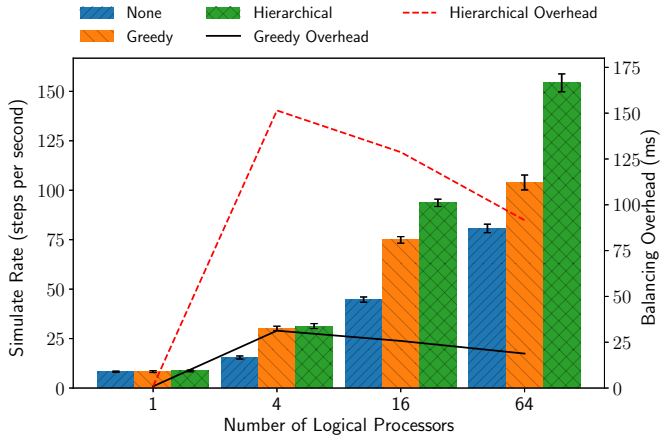

Fig. 13. Simulation Rate with various number of processors and load balancers for the DHeatBugs model.

the greedy balancer runs every 100 steps for simplicity. For the hierarchical balancer, the workload will be balanced every 100 steps at the lowest level, every 400 steps at the second lowest level, and every 1600 steps at the third lowest level, etc.

We test with the *DHeatBugs* application using a 6400× 6400 field and 3200 agents. To show the effectiveness of the load balancer, we start the simulation with the field manually partitioned in a very unbalanced way. Since the agents are uniformly and randomly placed in the field, with unbalanced partition, we expect a very unbalanced workload at each node.

Since the hierarchical load-balancer only works in a tree-like partition scheme while the greedy balancer works in any rectangularly partitioned field, we evaluate both partition schemes, one hierarchically partitioned with a quad tree, the other partitioned in a striped fashion, denoted as QT and SP, respectively. To demostrate the load balancers in action, we take a snapshot of a single run of the *DHeatBugs* with 16 processors and the two aforementioned initial partitions. Figure 12 shows the variance of the nodes' per-step runtimes as the simulation progresses. As we discovered from the experiments, with the striped partition the greedy balancer (SP-Greedy) works well in bringing down the variance of the runtime. With the hierarchical partition, the greedy balancer (QT-Greedy) is gradually stuck with a local optima and only moderately reduces the variance while the hierarchical balancer (QT-Hierarchical) is very effective in balancing the workload. As shown in the Figure, after two levels of load balancing, the variance of the nodes' runtime becomes nearly zero, indicating a well-balanced workload.

Next we evaluate the simulation rate and the balancing overhead of the two balancers with different number of logical processors. Figure 13 shows the simulation rate (bar plots using the left Y-axis) and the overhead of the balancer (lines using the right Y-axis) in DHeatBugs with a hierarchically partitioned field. Similar to the previous result, the hierarchical balancer can keep a balanced workload among nodes, resulting in the highest average simulation rate. Even though the greedy balancer performs poorly in balancing the workload, its overhead is much lower compared to the overhead of the hierarchical balancer. The overhead of both balancers decreases as the the number of processors increases. This is because with the same field size, the more processors the system uses, the smaller the region on each processor is and therefore less expensive for the balancer to rearrange the data between the processors.

VII. Conclusion

This paper described the design and implementation of Distributed MASON, a distributed version of the MASON multi-agent simulation toolkit. We first described Distributed MASON's local agent-based data management scheme using *Halo Exchanges* along with global data sharing via RPC calls. We then described our approach for achieving scalability and high performance using multiple types of load balancing algorithms. At the local level LPs manipulate their Areas-of-Interest in response to changes in workload distribution. At a global level Distributed MASON can use a centralized approach where runtime information is collected by a single LP. We considered several heuristics for load balancing, including a greedy approach and a tree-based hierarchical method. We then evaluated our system using an implemented running on an AWS C5 instance. We found that Distributed MASON achieved highly scalable performance, in terms of linear performance increases as the size of the simulation grew. We also found that its load balancing scheme was effective and was, not surprisingly, sensitive to the size and density of the workload.

Acknowledgments

This work is supported by the National Science Foundation under grant 1727303.

References

[1] C. Cioffi-Revilla, J. D. Rogers, P. Schopf, S. Luke, J. Bassett, A. Hailegiorgis, W. Kennedy, P. Froncek, M. Mulkerin, and M. Shaffer, "MASON NorthLands: A geospatial agent-based model of coupled human-artificial-natural systems in boreal and arctic regions," in *European Social Simulation Association (ESSA)*, 2015.

[2] K. Sullivan, E. Wei, B. Squires, D. Wicke, and S. Luke, "Training heterogeneous teams of robots," in *Autonomous Robots and Multirobot Systems (ARMS)*, 2015.

[3] V. Suryanarayanan, G. Theodoropoulos, and M. Lees, "PDES-MAS: Distributed simulation of multi-agent systems," *Procedia Computer Science*, vol. 18, pp. 671–681, 2013, 2013 International Conference on Computational Science.

[4] "SWARM agent-based simulation toolkit," http://www.swarm.org/.

[5] M. Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, 1994, system available at http://education.mit.edu/starlogo.

[6] "NetLogo simulation platform," http://ccl.northwestern.edu/netlogo/.

[7] M. Parker, Ascape multiagent simulation tool. http://ascape.sourceforge.net/, 1998.

[8] N. Collier, "Repast: An agent based modelling toolkit for java," 2001, system available at http://repast.sourceforge.net.

[9] "FLAME multiagent simulation tool," http://www.flame.ac.uk.

[10] N. Collier and M. North, *Large-Scale Computing Technologies for Complex System Simulations*. Wiley, 2012, ch. Repast HPC: A Platform for Large-Scale Agent-Based Modeling.

[11] R. D. Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo, "Bringing together efficiency and effectiveness in distributed simulations: the experience with D-Mason," *Simulation: Transactions of The Society for Modeling and Simulation International*, 2013.

[12] K. Sullivan, M. Coletti, and S. Luke, "GeoMason: Geospatial support for MASON," Available at http://cs.gmu.edu, Department of Computer Science, George Mason University, 4400 University Drive MSN 4A5, Fairfax, VA 22030-4444 USA, Tech. Rep. GMU-CS-TR-2010-16, 2010.

[13] W. Cai, S. J. Turner, and H. Zhao, "A load management system for running hla-based distributed simulations over the grid," in *Proceedings. Sixth IEEE International Workshop on Distributed Simulation and Real-Time Applications*, 2002, pp. 7–14.

[14] R. E. D. Grande and A. Boukerche, "Dynamic balancing of communication and computation load for HLA-based simulations on large-scale distributed systems," *J. Parallel Distrib. Comput.*, vol. 71, pp. 40–52, 2011.

[15] T. G. Alghamdi, R. E. De Grande, and A. Boukerche, "Enhancing load balancing efficiency based on migration delay for large-scale distributed simulations," in *Proceedings of the 19th International Symposium on Distributed Simulation and Real Time Applications*, ser. DS-RT 2015. Piscataway, NJ, USA: IEEE Press, 2015, pp. 33–40.

[16] N. Collier, J. Ozik, and C. M. Macal, "Large-scale agent-based modeling with repast hpc: A case study in parallelizing an agent-based model," in *Euro-Par 2015: Parallel Processing Workshops*, S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander, Eds. Springer International Publishing, 2015, pp. 454–465.

[17] A. Nouman, A. Anagnostou, and S. J. E. Taylor, "Developing a distributed agent-based and des simulation using portico and repast," in *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, Oct 2013, pp. 97–104.

[18] J. D. Westervelt, "Geographic information systems and agent-based modelling," in *Integrating Geographic Information Systems and Agent-Based Modelling Techniques for Simulating Social and Ecological Processes*, H. R. Gimblett, Ed. Oxford Univ. Press, 2002, pp. 83–104.

[19] I. Benenson and P. M. Torrens, *Geosimulation : Automata-Based Modelling of Urban Phenomena*. John Wiley and Sons, 2004.

[20] A. J. Heppenstall, A. T. Crooks, L. M. See, and M. Batty, Eds., *Agent-Based Models of Geographical Systems*. Springer, 2012.

[21] GAMA. GAMA modeling and simulation development environment. [Online]. Available: http://gama-platform.org/

[22] A. T. Crooks, A. Heppenstall, and N. Malleson, "Agent-based modelling," in *Comprehensive Geographic Information Systems*, B. Huang, Ed. Elsevier, 2018, pp. 218–243.

[23] R. Axtell, J. M. Epstein, J. S. Dean, G. J. Gumerman, A. C. Swedlund, J. Harburger, S. Chakravarty, R. Hammond, J. Parker, and M. Parker, "Population growth and collapse in a multiagent model of the kayenta anasazi in long house valley," *Proceedings of the National Academy of Sciences*, vol. 99, no. 3, pp. 7275–7279, 2002.

[24] A. Crooks and A. Hailegiorgis, "An agent-based modeling approach applied to the spread of cholera," *Environmental Modelling and Software*, vol. 62, pp. 164–177, 2014.

[25] N. Malleson, L. See, A. Evans, and A. Heppenstall, "Implementing comprehensive offender behaviour in a realistic agent-based model of burglary," *SIMULATION*, vol. 88, no. 1, pp. 50–71, 2012.

[26] B. Pires and A. Crooks, "Modeling the emergence of riots: A geosimulation approach," *Computers, Environment and Urban Systems*, vol. 61, pp. 66–80, 2017.

[27] W. G. Kennedy, C. R. Cotla, T. R. Gulden, M. Coletti, and C. Cioffi-Revilla, "Towards validating a model of households and societies of east africa," in *Advances in Computational Social Science: The Fourth World Congress*, S. H. Chen, I. Terano, H. Yamamoto, and C. C. Tai, Eds. New York, NY: Springer, 2014, pp. 315–328.

[28] C. W. Reynolds, "Flocks, herds, and schools: a distributed behavioral model," *Computer Graphics*, vol. 21, no. 4, pp. 25–34, 1987, sIGGRAPH '87 Proceedings.

[29] "JTS topology suite," https://github.com/locationtech/jts.

[30] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.