Proceedings of the 2013 Winter Simulation Conference
R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, eds.

MULTITHREADED AGENT-BASED SIMULATION

Michael E. Goldsby
Carmen M. Pancerella

7011 East Avenue
Sandia National Laboratories
Livermore, CA 94550, USA

ABSTRACT

Multithreading can significantly increase the performance of large agent-based simulations on multicore systems, but agent-based software packages do not commonly offer adequate support for multithreading. This report describes alterations and additions made to the MASON agent-based simulation package that allow the application programmer to make use of multiple threads easily and without radical change to conventional agent-based programming style. The report confirms performance gains with the results of test runs.

1 INTRODUCTION

The work reported here grew out of a project that needed the results of simulation runs with a minimum of 100,000 agents (Armstrong et al. 2011). A shared-memory multiprocessor server was available, and we realized that we could speed up the runs if we could use multiple processors in each run, but we could find no agent-based simulation package with adequate support for multithreading. Some packages would allow the programmer to use multiple threads, but the data structures were not thread-safe, and they left all issues of division of work and avoidance of conflicting accesses up to the application programmer.

This report describes alterations and additions to the MASON (Luke et al. 2005) agent-based simulation package that allow the application programmer to use threads with ease. We realized at the beginning that widespread use of locks would have been complicated and an almost sure-fire performance killer. As it turned out, we devised a scheme requiring no locks at all. The altered system shows good performance gains in the test runs.

The next section briefly describes agent-based simulation. Section 3 describes MASON, the agent-based simulation package chosen for this work. Section 4 describes the basic approach. Section 5 gives details of the implementation. Section 6 describes certain coding conventions required of the application. Section 7 comments on related work. Section 8 reports the results of performance tests with the altered system. Finally, section 9 contains conclusions and suggestions for future work.

2 AGENT-BASED SIMULATION

In agent-based modeling (ABM), a system is modeled as a collection of autonomous decision-making entities (agents) that encapsulate the behavior of individuals in a system. Each agent individually assesses its state, makes decisions based on rules, and interacts with other agents. Collectively the agents emulate a real-world system. Multi-agent systems (MAS) or agent-based simulation (ABS) is the corresponding computational technique, where a software agent executes on a processor and communicates with other agents in the system, in order to model a complex system; see Epstein and Axtell (1996), Jennings (2000), and Uhrmacher and Weyns (2009) for more detail. Each agent maintains its own state, and there is no centralized control.

Agent-based simulation has become an important method of modeling not only how individuals behave but also how the behavior and interactions of individuals affects the larger system. Thus, ABS is used to model socio-economic systems in economics, finance, supply chain, traffic modeling, social networking, and others.

There are a number of ABM software platforms that can be used to develop agent-based simulations (Railsback, Lytinen, and Jackson 2006). Two such agent-based platforms are Repast (North, Collier, and Vos 2006) and MASON, both implemented in Java. Repast is arguably the most popular agent-based simulation package in use and has active support. MASON is a much smaller, less mature, and leaner package, which nevertheless implements the core constructs needed for agent-based simulation. MASON was designed with execution speed a priority.

Such agent-based systems are typically single-threaded, with support for parallelism rudimentary or lacking. Both Repast and MASON have a minimal level of support for multithreading, but in neither are the data structures thread-safe, nor is there support for partitioning data among the threads. An exception is the high-performance computing version of Repast, written in C++, which uses MPI (MPI Forum 2013) as the underlying communication layer and runs a single user thread in each MPI process.

3 MASON

The agent-based system chosen for this work is MASON. MASON is a full-featured simulation package with a relatively small and well-commented code base. We chose MASON over Repast, which is more widely used than MASON and has features that are roughly a superset of MASON's, because MASON's codebase is smaller and easier to adapt.

MASON provides several forms of shared data structures or *fields* for the environment: 2- and 3-dimensional discrete grids, which have both dense and sparse forms, 2- and 3-dimensional continuous spaces, and networks. There are three types of discrete dense grids distinguished by their values, integer, floating point (double) and single arbitrary object. Sparse grids can contain an arbitrary collection of objects at any grid point, and continuous spaces, which are backed by sparse grids in implementation, can associate an arbitrary collection of objects with any location within a rectangular area or volume.

MASON also includes network fields, which define nodes connected by edges and allow the construction of arbitrary graphs. We have not included networks in the present implementation, but we believe it is possible to extend our scheme to them.

MASON has no distinct agent type. Any object implementing the `Steppable` interface, which defines the `step` method, may play the role of an agent. Although agents typically interact through the environment, one agent may call the methods of another directly if it has a reference to it; however, we will place certain restrictions on this form of interaction, as explained below.

MASON already contains low-level support for parallelism. It provides a `Steppable` class named `ParallelSequence` that contains an array of threads and fans a single `step()` call out into `step()` calls by each of the threads on the respective members of an array of user-provided `Steppable` objects. If a program uses the single-program multiple-data (SPMD) paradigm, each `Steppable` in the array is the same except for parameterization. The threads synchronize at each step boundary by means of a semaphore class defined in MASON. MASON leaves all other issues involving synchronization, concurrent access to data and division of work in the hands of the user, however.

4 APPROACH

The approach is intuitive and leaves almost all of the existing package intact. We decompose the area or volume of each field into as nearly as possible equal non-overlapping parts, which we call partitions, and assign a thread to each partition. The thread is responsible for processing the parts of each field that belong to its partition. The intention is to use one thread per available hardware processor. Figure 1 shows a 200 by 200 field divided into four partitions.

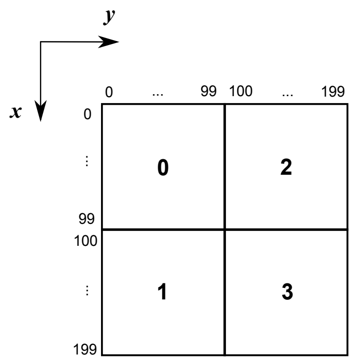

Figure 1: A 200x200 field with four partitions.

To resolve the issues of thread safety and conflicting data accesses, we use a combination of barrier synchronization and coding convention. We set the convention that a partition's thread may write only to locations (in a grid or space) belonging to the partition. However, it is often necessary for a partition's thread to read values from a neighboring partition. In order to avoid reading a value while the owning partition's thread is writing it, we divide each simulation step into two parts separated by a barrier synchronization. In the first part, a partition's thread may not write to any field (more precisely, it may not write any value that may be read by another partition's thread), but it may read from anywhere in a field, including parts of the field belonging to other partitions. During the second part of the step, a thread may read and write any value in its own partition. See section 6 for a more detailed explanation.

It is also often necessary for an agent to move from a part of a field belonging to one partition to a part belonging to another. We accomplish such movement transparently to the application programmer by means of interprocessor message queues. See section 5 for details.

Our approach allows code from already-existing application to run in parallel within our framework with relatively small changes (see example code in the appendices).

5 IMPLEMENTATION

This section describes the changes and additions to the MASON codebase.

MASON has discrete-event scheduling (for a single host), but most agent-based simulation uses simple time-stepping (as indeed the name of the `step` method suggests.) Hence we gear our implementation toward time-stepped simulations. In writing the `Steppable` objects, we found it far more efficient to schedule a single call to the `step` method per timestep and then to call the `step` methods of the agents iteratively than to schedule `step` calls for each agent.

MASON has a graphical user interface that provides a visual representation of the actions of the agents in the fields, but we made no attempt to preserve it in the multithreaded version, which we see as being most suitable for large batch runs.

In all, the implementation required changes to 13 MASON modules of the 104 total MASON modules, exclusive of the display and demo code. (A module is a compilation unit: a separate file containing Java source code and having file extension ".java".) The changes were not rewrites but additions to already existing functionality. To that we added 16 modules of our own. Many of these are short and simple (an interpartition message class, nine classes associated with reductions, the `Transportable` interface described below). There are four modules to compute the extents of the 2D or 3D grids or spaces owned by each partition.

5.1 Interpartition Messages

Many applications allow agents to move in their environment (for instance, the two test programs described in Section 8). In the course of such movement, an agent may cross from one partition into another. The usual field of play for agents is the sparse field (sparse grid or continuous space), since it contains the machinery to allow multiple agents to share a location. The sparse field method `setObjectLocation` allows the program to move an agent from one position in the field to another. We added logic to the `setObjectLocation` method that sends the agent to a neighboring partition wrapped in an interpartition message if the destination location lies in a different partition than the agent already occupies. Thus transition from one partition to another is handled automatically by the run-time system.

The application may need to make its own adjustments when an agent passes from one partition to another. For instance, if each partition's logic keeps track of the agents in that partition, the source partition may need to remove the agent from its collection when it leaves, and the destination partition may need to add the agent to its collection when it arrives. For that reason, we define the `Transportable` interface, with methods `transporting` and `transported`. Any agent that may travel from one partition to another must implement this interface. When the `setObjectLocation` method learns that the object is leaving a partition, it calls the `transporting` method on the object, and when the run-time system delivers the message, it calls the object's `transported` method.

To support interpartition messaging, we define an interpartition message queue for each partition (using class `java.util.ConcurrentLinkedQueue`) and provide a statically available method for sending interpartition messages. The run-time system delivers interpartition messages immediately after the start of a simulation step. The synchronization performed by MASON at the beginning of each step ensures that the partitions have sent all interpartition messages for the previous step. It is also necessary to synchronize after delivery of the messages, since the delivery may alter a field value later read by a neighboring partition. We include a static `synchronize` method, which the run-time system calls for this purpose; the method synchronizes all partitions at a barrier using `java.util.concurrent.CyclicBarrier`.

5.2 Fields

The implementation shares each MASON field among all the partitions, so there is a single data structure for each field on which all the partitions threads operate.

The sparse fields in MASON maintain common data structures for the entire field, so two partition threads that are trying to write to their respective parts of the field could nonetheless suffer a conflict. In order to avoid locks and bottlenecks, we multiplexed these data structures over the partitions, replacing them with an array of structures, giving each partition access to one element of each array. Doing so adds the overhead of an access to a map (`java.util.concurrent.ConcurrentHashMap`) to select the correct element on every access to the sparse field. This was the most invasive of all the changes, although it was mechanically a simple one.

The dense grids provide a number of global actions (`sum`, `min`, `max`, `mean`, etc.). The partitioned system uses reductions to implement these actions. Support for the reductions includes a reduction message queue for each partition, implemented using `java.util.concurrent.LinkedBlockingQueue`. The reductions are invisible to the application programmer.

5.3 Thread-local State

The new `ThreadState` module uses a java `ThreadLocal` variable to contain information pertaining to each individual partition. The `ThreadState` class contains the partition identifier (an integer in the range 0 to $np-1$, where np = the number of partitions) and a reference to a random number generator. We give each partition its own random number generator; users may force the partitions to use any seed values they want, but by default the system derives a different seed for each partition from the current time.

The programmer may extend the `ThreadState` object to contain any information that is local to a partition for a particular application. In the original MASON system, the `SimState` object contained the entire state of the simulation; in our partitioned version, the simulation state is spread over the `SimState` object and the `ThreadState` object of each partition.

6 CODING CONVENTIONS

Here we describe in more detail the coding conventions mentioned above. We make the convention that a partition's thread may write only to locations (in the grid or space) belonging to the partition. It is frequently necessary for a partition's thread to read values from a neighboring partition (for instance, if the value at a grid point on the border of the partition depends on the values at the surrounding grid points). In order to avoid reading a value while the owning partition's thread is writing it, we set the convention that each simulation step be divided into two parts. During the first part, a partition's thread may not write to a field (or more precisely, it may not write any value that may be read by another partition's thread), but it may read from anywhere in a field, even from parts of the field belonging to other partitions. During the second part, a partition may read and write any value in its own partition. If an agent directly calls a method of another agent, it must do so in accordance with these conventions; that is, the call must not cause a write to another partition or a write to the same partition during the read part of the step.

Note that it is the user's responsibility to abide by the convention that the simulation step be divided into a read part and a write part; there is no enforcement by the run-time system or the compiler. The static `synchronize` method described in section 5.1 is also available to the application, which must call it at the end of the read part of the step in order to guarantee that no cross-partition reads are in progress when the application writes a field value. It is necessary to observe this convention to ensure correctness and avoid exceptions (in particular `java.util.ConcurrentModificationException`).

Separating the reading and writing parts of a simulation step also lays the foundation for ensuring that the runs are repeatable in the sense that two runs done with the same random number generator seeds and the same number of partitions must produce exactly the same results.

Figure 2 represents a single timestep, initiated by MASON's synchronization at a semaphore. The run-time does a barrier synchronization after delivery of the interpartition messages, and the user (by convention) does a barrier synchronization before the write part of the step.

Appendices A and B show the code for the `step` method of the original MASON Flockers demo program and the `step` method used in our partitioned version, respectively. Note that the statements they contain are nearly identical, and the two differ only in that the altered version uses a thread-local random number generator and segregates the statements according to whether they execute in the read part or the write part of a timestep. The statements that differ between the two are marked with an asterisk in the first column in Appendix B.

7 RELATED WORK

Voss et al. (2010) describe an approach for the Repast system similar to that followed here for MASON. As in the work described here, it uses partitioned data with multiple threads. It does not explicitly describe cross-partition access or travel, but mentions that synchronized blocks are used for thread safety when necessary. The work presented here avoids the overhead of synchronized blocks by using barrier synchronization to rule out conflicting accesses and transparently uses lock-free message queues for interpartition travel by agents. They note, as we do, that iterating over the agents sustains far less overhead than scheduling each agent action.

Armstrong et al. (2011) present an earlier version of the work described here. That version gave each partition separate data structures, and the run-time system performed a translation from a global to a local field location for each field access. It maintained shadow copies in each partition of the nearby elements of neighboring partitions, which were the only elements outside its partition that a partition's thread could

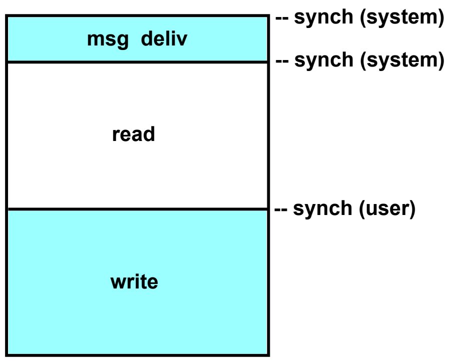

Figure 2: A single timestep.

read. There was, however, no requirement for special coding conventions that divided the time step into read and write portions, although the user might elect to use such a division to achieve repeatability. The present version requires fewer and simpler changes and shows better performance than the earlier version.

8 PERFORMANCE

The platform used for performance runs was a shared memory machine with 32 Intel Xeon 2.27 GHz processors, cache size 24576 kB and a 64GB memory space. All processors remained enabled during all tests.

The performance tests used two demo program distributed with the MASON package, Flockers and HeatBugs. Flockers is similar to Craig Reynolds' Boids (Reynolds 1987) and simulates the flocking of birds in flight. The agents fly in a 2-dimensional continuous space. The agents in HeatBugs exchange heat with their environment and seek an ideal temperature. If an agent (bug) is above (below) that temperature, it moves to the coldest (hottest) location in its immediate (Moore) neighborhood. HeatBugs operates in two alternating phases: in the first phase the bugs move, and in the second the environment (represented by a dense floating point grid) diffuses its heat by allowing conduction from a cell's Moore neighborhood to or from the cell depending on the temperature difference and a specified conduction rate. The nature of both programs is such that their computational complexity increases linearly with problem size.

Both test programs are canonical agent-based simulations representing swarm models, swarming being an emergent behavior from the local rules followed by individual agent. These test programs have significance to the types of problems that are being modeling in multi-agent simulations. For example, the self-organizing behavior of flocking in an agent-based simulation has been used to study how leaders emerge in a group as a consequence of the interactions of the members (Quera, Beltran, and Dolado 2010), and a heat bugs model was adapted to model pollution in a metropolitan city (Ahat et al. 2009).

In addition, the patterns of computation followed by the two programs can apply to a much larger class of models. The Flockers program iterates through the agents and updates each according to information gained from the other agents in its immediate neighborhood. The HeatBugs program iterates through the agents and updates each according to information gained from the environment in its immediate vicinity, then iterates through the locations of the environment and alters the information at each location according to information gained from the surrounding locations. These rather general patterns suggest that the performance results may be indicative of performance over a larger class of applications.

We use three different problem sizes each for HeatBugs and Flockers. The density of agents per field area is the same for all three sizes for HeatBugs and similarly for Flockers. The problem size remains the

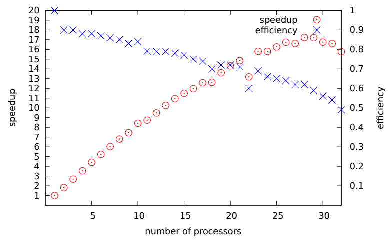

Figure 3: HeatBugs: 1600000 agents on 12649x12649 field.

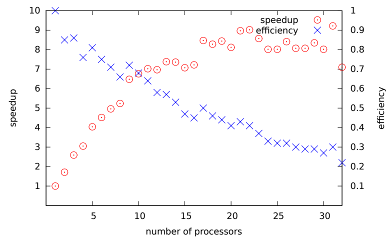

Figure 4: HeatBugs: 100000 agents on 3162x3162 field.

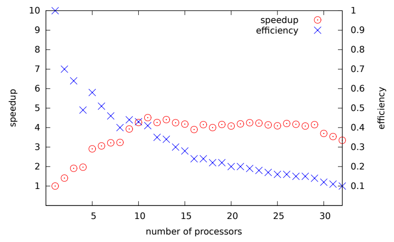

Figure 5: HeatBugs: 6250 agents on 791x791 field.

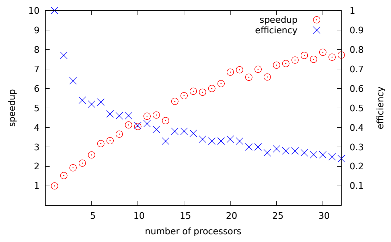

Figure 6: Flockers: 1280000 agents on 12649x12649 field.

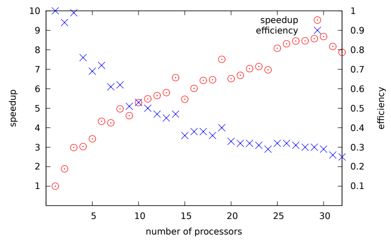

Figure 7: Flockers: 80000 agents on 3162x3162 field.

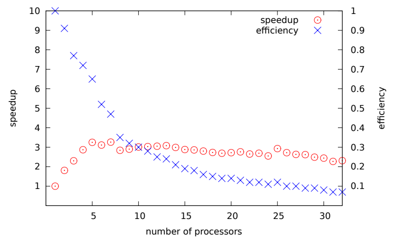

Figure 8: Flockers: 5000 agents on 791x791 field.

same in each case as the number of processors increases (strong scaling). The test programs never used more than about 16GB of memory.

Figures 3 through 8 show the results of the performance runs. They show both speedup and efficiency versus the number of partitions p, where speedup is the time of one simulation step performed with one partition divided by the time of one simulation step performed with p partitions, and efficiency is speedup divided by p. We show the average of five runs for each configuration except the two largest (Flockers with 1,280,000 agents and HeatBugs with 1,600,000 agents), for which we used the average of three runs.

Scaling varies with the problem and the problem size, as Figures 3 through 8 show. HeatBugs with 1,600,000 agents scales extremely well up to about 10 processors (speedup of 8.42 with 10 processors), scales moderately well up to 20 processors (speedup of 14.3 with 20 processors) and does not flatten out entirely until it reaches 28 processors. HeatBugs with 100,000 agents scales very well up to three processors (speedup of 2.6 with with three processors) and moderately well up to about 10 processors (speedup of 6.8 with 10 processors.) The scaling of HeatBugs with 6,250 agents is not impressive and reaches its maximum speedup of 4.5 at 11 processors. These results show that for good scaling, it is important to have a large enough problem over which to amortize the overhead of parallelization.

The largest Flockers problem size scales about half as well as the largest HeatBugs size. The mid-size Flockers runs scale nearly as well as HeatBugs up to five or six processors and then falls off somewhat more rapidly. The smallest Flockers scales about as well as the smallest HeatBugs up to five processors but tapers off rapidly after that. We suspect that the tendency of the agents to move together in flocks, leading to concentrations of agents in some parts of the field and absence of agents in other parts, may have caused processor loads to become unbalanced; this hypothesis will have to be verified with additional tests.

9 CONCLUSIONS

The work presented here shows that it is not overly difficult to add multithreading to an agent-based simulation system with a clearly written codebase, and that multithreading on multicore hardware can significantly decrease the run time of large agent-based simulation runs. Moreover, multithreading can be accomplished in such a way that the code written for the multithreaded system closely resembles code for a conventional single-threaded system.

Our performance results on two different agent-based simulation problems indicate that other large agent-based simulations might also benefit from multithreading. Additional performance studies would be useful to confirm the advantages of the proposed approach.

An additional performance enhancement would be to do load balancing. It would not be difficult to keep track of the amount of time each partition's thread spends waiting for synchronization and periodically adjust the size of the partitions to distribute the waiting more evenly over them. As yet we have no data on how load balancing would affect performance; obviously it would have significant effect only on applications in which the load tends to become unbalanced because of agent movement or other factors.

A logical next step would be to extend the scheme to multiple network-connected hosts by distributing each grid or space over all the hosts, each implementing a section of it. Interhost communication latency would be a critical factor for performance. One means of diminishing the effect of latency is to trade frequency of communication for duplication of data and processing. By restricting the depth to which a read access can penetrate into a neighboring partition and the depth to which an agent can travel into a neighboring partition in a single simulation step, and by duplicating the contents and the processing of the border regions of a partition in the neighboring partitions, we could decrease the frequency of interhost communication and amortize the communication latency over several simulation steps (Aaby, Perumalla, and Seal 2010) .

ACKNOWLEDGMENTS

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U. S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

A ORIGINAL MASON CODE

```
public void step(SimState state)
    {
    final Flockers flock = (Flockers)state;
    loc = flock.flockers.getObjectLocation(this);

    if (dead) return;

    Bag b = getNeighbors();

    Double2D avoid = avoidance(b,flock.flockers);
    Double2D cohe = cohesion(b,flock.flockers);
    Double2D rand = randomness(flock.random);
    Double2D cons = consistency(b,flock.flockers);
    Double2D mome = momentum();

    double dx = flock.cohesion * cohe.x
                 + flock.avoidance * avoid.x
                  + flock.consistency* cons.x
                   + flock.randomness * rand.x
                    + flock.momentum * mome.x;
    double dy = flock.cohesion * cohe.y
                 + flock.avoidance * avoid.y
```

```
                    + flock.consistency* cons.y
                     + flock.randomness * rand.y
                      + flock.momentum * mome.y;

        double dis = Math.sqrt(dx*dx+dy*dy);
        if (dis>0)
            {
            dx = dx / dis * flock.jump;
            dy = dy / dis * flock.jump;
            }

        lastd = new Double2D(dx,dy);
        loc = new Double2D(flock.flockers.stx(loc.x + dx),
                           flock.flockers.sty(loc.y + dy));
        flock.flockers.setObjectLocation(this, loc);
        }
```

B PARTITIONED CODE

```
*   public void step_r(SimState state, Flockers.LocalState local)
        {
        final Flockers flock = (Flockers)state;
        loc = flock.flockers.getObjectLocation(this);

        if (dead) return;

        Bag b = getNeighbors();

        Double2D avoid = avoidance(b,flock.flockers);
        Double2D cohe = cohesion(b,flock.flockers);
*       Double2D rand = randomness(local.random());
        Double2D cons = consistency(b,flock.flockers);
        Double2D mome = momentum();

        double dx = flock.cohesion * cohe.x
                   + flock.avoidance * avoid.x
                    + flock.consistency* cons.x
                     + flock.randomness * rand.x
                      + flock.momentum * mome.x;
        double dy = flock.cohesion * cohe.y
                   + flock.avoidance * avoid.y
                    + flock.consistency* cons.y
                     + flock.randomness * rand.y
                      + flock.momentum * mome.y;

        double dis = Math.sqrt(dx*dx+dy*dy);
        if (dis>0)
            {
            dx = dx / dis * flock.jump;
            dy = dy / dis * flock.jump;
```

```
              }

         lastd = new Double2D(dx,dy);
         loc = new Double2D(flock.flockers.stx(loc.x + dx),
                            flock.flockers.sty(loc.y + dy));
*        }
*    public void step_w(SimState state, Flockers.LocalState local)
*        {
*        final Flockers flock = (Flockers)state;
         flock.flockers.setObjectLocation(this, loc);
         }
```

An asterisk in the first column marks the lines of code that differ from the original code in Appendix A.

REFERENCES

Aaby, B. G., K. S. Perumalla, and S. K. Seal. 2010. "Efficient Simulation of Agent-Based Models on Multi-GPU and Multi-Core Clusters". In *3rd International ICST Conference on Simulation Tools and Techniques*, edited by L. F. Perrone, G. Stea, J. Liu, A. Uhrmacher, and M. Villén-Altamirano. Institute for Computer Sciences, Social Informatics and Telecommunications Engineering: ICST/ACM.

Ahat, M., S. B. Amor, M. Bui, M. Lamure, and M.-F. Courel. 2009. "Pollution Modeling and Simulation with Multi-Agent and Pretopology". In *Complex Sciences: First International Conference, Complex 2009*, edited by J. Zhou, 225–231. Institute for Computer Sciences, Social Informatics and Telecommunications Engineering: Springer.

Armstrong, R. C., J. S. Schoeniger, K. L. Sale, M. Goldsby, and J. Mayo. 2011. "Community-Based Resistance to Intrusion in Information Technology Systems". Sandia report SAND2011-8839, Sandia National Laboratories.

Epstein, J. M., and R. Axtell. 1996. *Growing Artificial Societies*. Cambridge, MA, USA: Bradford.

Jennings, N. R. 2000. "On Agent-Based Software Engineering". *Artificial Intelligence* 117 (2): 277–296.

Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. C. Balan. 2005. "MASON: A Multiagent Simulation Environment". *Simulation* 81 (7): 517–527.

MPI Forum 2013. "The Message Passing Interface (MPI) Standard". Accessed Mar. 3, 2013. http://www.mcs.anl.gov/research/projects/mpi/.

North, M. J., N. T. Collier, and J. R. Vos. 2006. "Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit". *ACM Transactions on Modeling and Computer Simulation* 16 (1): 1–25.

Quera, S., F. S. Beltran, and R. Dolado. 2010. "Flocking Behavior: Agent-Based Simulation and Hierarchical Leadership". *Journal of Artificial Societies and Social Simulation* 13 (2): 8.

Railsback, S. F., S. L. Lytinen, and S. K. Jackson. 2006. "Agent-Based Simulation Platforms: Review and Development Recommendations". *SIMULATION* 82:609–623.

Reynolds, C. 1987. "Flocks, Herds and Schools: A Distributed Behavioral Model". In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, edited by M. C. Stone, 25–34. Association for Computing Machinery: ACM.

Uhrmacher, A. M., and D. Weyns. 2009. *Multi-Agent Systems: Simulation and Applications*. Boca Raton, FL, USA: CRC Press.

Voss, A., J.-Y. You, E. Yen, H.-Y. Chen, S. Lin, A. Turner, and J.-P. Lin. 2010. "Scalable Social Simulation: Investigating Population-Scale Phenomena Using Commodity Computing". In *Proceedings of the IEEE 6th Conference on e-Science*. Brisbane, Queensland, Australia: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

MICHAEL E. GOLDSBY is a contractor to Sandia National Laboratories, California, with special interests in concurrent and distributed systems and simulation. He holds a B.Sc. in Mathematics from the University of Arizona and is a member of ACM. His email address is michaelegoldsby@gmail.com and his web page is http://michaelegoldsby.com.

CARMEN M. PANCERELLA is a Principal Member of the Technical Staff at Sandia National Laboratories in Livermore, California. She holds a M.S. and Ph.D. In Computer Science from the University of Virginia. Over the years, her modeling and simulation research interests have included parallel discrete event simulation, agent-based modeling, and the use of models in support of emergency preparedness. Her email address is carmen@sandia.gov.