Proceedings of the 2011 Winter Simulation Conference
S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, eds.

ON-THE-FLY PARALLELIZATION IN AGENT-BASED SIMULATION SYSTEMS

Cole Sherer

Computer Science Department
University of Georgia
Athens, GA 30602, USA

George Vulov

CSE, College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280, USA

Maria Hybinette

Computer Science Department
University of Georgia
Athens, GA 30602, USA

ABSTRACT

Agent-**b**ased **s**imulation (ABS) systems are increasingly being used to solve a wide-array of problems in business, telecommunications, robotics, games, and military applications. ABS modelers face two challenges: First, performance is affected, as their simulations become more complex and larger scale; and second, development is difficult because there is no common interface to the array of platforms that support ABS work. We seek to transform popular, intuitive, sequential ABS APIs into efficient parallel code automatically. As a first step we are parallelizing the popular MASON multiagent simulation kit, other future potential targets include Player/Stage and Teambots. To achieve this, we have mapped the core MASON API to correlate with the agent API of SASSY, a parallel and scalable, agent-based simulation system. We then use Soot, a Java bytecode optimization framework, to automatically convert MASON bytecode into SASSY bytecode. This allows simple, sequential MASON code to be run in a parallel environment.

1 INTRODUCTION

Agent-**b**ased **s**imulation (ABS) systems are increasingly being used to solve a wide-array of problems in business, telecommunications, robotics, computer games, and military applications (Logan and Theodoropoulos 2001). With these applications, ABS modelers face two challenges: first, performance is affected as their simulations become more complex and larger scale, and second, development is difficult because there is no common interface to the array of platforms that support ABS work. In this paper we seek to address the first issue. By providing a framework to seamlessly translate sequential ABS bytecode to parallel, distributed bytecode, we can tackle larger scale simulations in a smaller amount of time.

Many compilers have been designed to tackle the general auto-parallelization problem, but they perform poorly. Most rely on a developer to provide annotations to aid in the parallelization process, and others use an optimistic parallelization process that may be *slower* than the serial version due to excessive rollback (Kulkarni et al. 2009). By focusing on ABS systems alone, we are able to solve this problem and provide proof of concept using the MASON framework as a starting point. Our method requires no annotations and can be scaled to any number of processors after being run through a transformer only once.

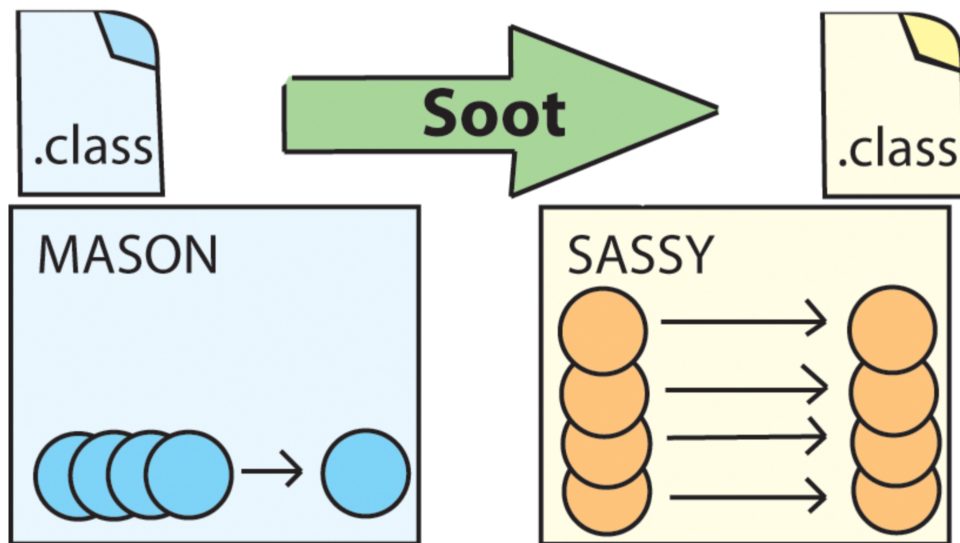

Figure 1: MASON bytecode is transformed into SASSY bytecode.

We seek to transform popular, intuitive, sequential ABS APIs into more efficient PDES code automatically. As a first step we are parallelizing the popular MASON multiagent simulation kit developed at George Mason University. Future potential targets include Player/Stage and Teambots. To achieve this, we have mapped the core of the MASON API to correlate with the agent API of SASSY, a scalable, agent-based simulation system using a PDES kernel. We then use Soot, a Java bytecode optimization framework, to automatically convert MASON bytecode into SASSY bytecode. This allows simple, sequential MASON code to be run in a PDES environment. This process is demonstrated in Figure 1. Our initial tests show promising results with an average speedup of 3.44 when run on 4 machines (3.44 times faster than the original running sequential code).

The remainder of this paper is organized as follows: in the next section we review related work from the high performance computing and agent-based simulation communities. In Section 3 we present our implementation in detail and discuss two simulations that were used in developing our approach. In Section 4 we discuss the results of our tests, comparing a MASON simulation to both a hand-coded SASSY simulation and one transformed using our Soot approach. Finally, we conclude the paper with a discussion of future work.

2 RELATED WORK

MASON is an easily extensible, discrete-event multi-agent simulation system written in Java. It is designed for running swarm, multi-agent simulations with millions of agents, but was specifically designed to run in a single process. This allows researchers to run many tests in parallel, using different parameters for each individual simulation. This is useful for genetic algorithm-based simulations where the parameters must evolve slowly until an optimum set is found for the problem (Luke et al. 2004). MASON can be extended such that agents in a single simulation are performing tasks in a parallel or distributed environment, but the onus is on the simulation developer to handle synchronization issues that may arise. We chose to target MASON due to its very active community and large number of recent publications.

Player/Stage is a multi-robot simulation system. Player is used as a robot device server offering transparent robot control over a network, and Stage is a lightweight, highly configurable simulator supporting large populations of agents. Player/Stage is socket-based, so it works with a wide range of languages from

C and Fortran to Java and Python. The system is most often limited by TCP latency and overhead, but it could potentially benefit from our technique (Gerkey et al. 2003). Teambots is also heavily rooted in robotics research and could benefit from our technique as well (Balch 1998).

SASSY is an agent-based simulation system built on a traditional PDES kernel. SASSY's kernel is based on the Time-Warp synchronization algorithm, and uses middleware to provide an agent-based API to application developers. This API is based on a sense-think-act cycle and logically separates an agent's brain from its body (Hybinette et al. 2006). Experiments in (Vulov et al. 2008) show that SASSY can achieve significant speedup (6.2x) when run on 8 processing elements (PEs) with thousands of agents.

General HPC parallelization approaches rely on two main approaches, static and runtime. Runtime techniques rely on optimistic parallelization. This method attempts to parallelize code, using rollbacks when necessary to prevent synchronization issues. When used on its own, optimistic parallelization can actually increase runtimes when there are excessive rollbacks. This type of auto-parallelization often works best when paired with static techniques like those of the Galois project. These rely on programming abstractions to highlight opportunities for parallelism. One of the main abstractions in the Galois project is a custom iterator that accesses items in a list in random order, using a barrier for synchronization at the end of iteration. This can be used with data structures that are semantically commutative, like sets, and signals the runtime analyzer that parallelization will work well over the iterated set (Kulkarni et al. 2009). SASSY also uses optimistic parallelization, but our static analysis technique does not rely on any special annotations from the developer.

While no general approach has been found to solve the auto-parallelization problem, we believe that by focusing on a subset of programs (namely multi-agent simulations written in MASON), we can easily produce parallel, distributed code using Soot, a framework designed for optimizing Java code. While Java has many advantages like platform independence, execution safety, garbage collection and an object-oriented paradigm, it is often much slower than C and C++. Soot was designed to optimize and annotate Java bytecode to remove unnecessary safety checks (like many array bounds checks), inline functions, and perform other tasks to get Java up to speed. Soot provides facilities for both intraprocedural and whole program optimization, making it perfect for transforming bytecode from one simulation library to another (Vallée-Rai et al. 1999).

3 APPROACH

Soot offers a fairly straightforward approach to parallelization. The first step in converting code from MASON into SASSY is to study the API of each framework and map the methods and classes from MASON into those of SASSY. A brief overview of this mapping can be found in Figure 2. All MASON simulations contain exactly one object that extends the `SimState` class. This object contains a discrete-event schedule, a MersenneTwister random number generator, and zero or more fields. SASSY is designed for every agent to keep track of a local state and share relevant information with other agents. Since there is no world state kept in SASSY, the `SimState` class posed our first major challenge. In order to mitigate this a `WorldStateAgent` (WSA) class extending SASSY's Agent class is created to take the place of MASON's `SimState`. This WSA would have to send messages to all other agents at each time-step informing them of the current world state. This imposes memory usage issues that will be discussed in Section 5.

Two important methods in the `SimState` class that need to be handled by SASSY are `start` and `finish`. We handle the `start` method by simply renaming it `init`, but the `finish` method is a bit more difficult. SASSY uses the `setEndOfSimFlag` method to signify that a simulation should be finished, so we must replace any calls to the `finish` method with calls to `setEndOfSimFlag` instead. If the `SimState` subclass overwrites `finish`, we only replace calls to `super.finish` to ensure that

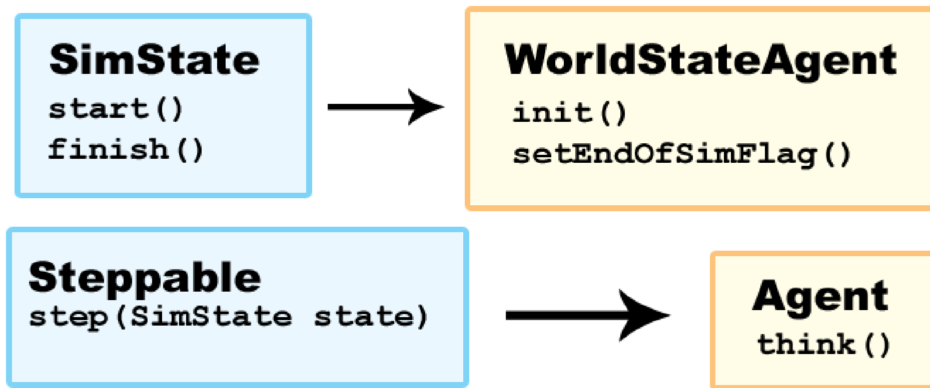

Figure 2: This is a mapping of MASON classes to SASSY classes. MASON is on the left; SASSY is on the right.

all custom cleanup code is run before the simulation is complete.

In MASON, the `SimState` object contains a schedule which is an ordered list of `Steppables`. As the schedule is stepped, the next scheduled agent is popped from the list and its `step` method is called. We replace all subclasses of `Steppable` with SASSY `Agents`, renaming the `step` method to `think`, fitting SASSY's sense-think-act model.

Our first experiment tested the system in a "stateless" environment. The simulation involves 100 balls bouncing around the screen. The balls ignore each other and can only interact with the boundary of the simulation. Each time step, a ball will calculate whether it is currently touching a boundary. If so, the ball will change directions to simulate a bounce. Balls all sleep from 1-3ms per `think` cycle to simulate an agent with more advanced logic. Each ball in the simulation advances 1000 time steps before signaling for the simulation to end. We first implemented this simulation in MASON and then by hand in SASSY. After running tests to confirm that SASSY could indeed perform faster than MASON when run in a distributed environment, we ran the MASON code through our Soot transformer and measured the performance of the auto-generated code. An image of this simulation can be seen in Figure 3.

Our second experiment was more complicated and was designed to really test the correctness of our bytecode transformer. For this experiment, we chose to use the ant foraging simulation that comes with the MASON library (Panait and Luke 2004). In this simulation, 20 ants search for food in a 100x100 grid. The ants use two pheromones: `findFood` and `findNest`, to mark their paths as they move around the world. These pheromones are slowly diffused through the grid to simulate degradation over time. Other ants in the simulation follow and add to these pheromones to locate their goals. Over time, this creates short, efficient paths between each goal in the simulation. This is normally run under visualization to view these paths, but in order to compare performance we turned all visualizations off and modified the simulation to end after 100 food was collected. This simulation involves keeping track of a lot of world state. As discussed earlier, this uses a lot of memory in an auto-parallelized version, where each ant keeps a copy of the entire world state. Our hand-coded version was designed to alleviate this memory usage problem.

In the hand-coded version of the ant foraging simulation, the shared environment was divided and each part of the environment was modeled by an individual agent. The shared ant environment has three essential functions:

1. Storing the levels of `findFood` and `findNest` pheromones

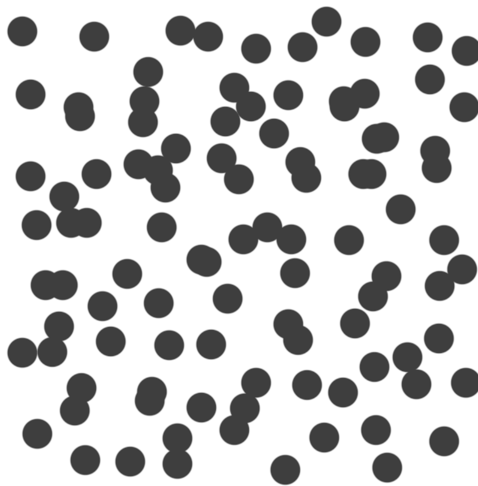

Figure 3: Simple ball simulation with 100 bouncing balls.

2. Modeling diffusion of each of the two pheromones
3. Modeling evaporation of the two pheromones.

While diffusion is an inherently continuous process, its implementation in a computer system requires discretization. We discretized the environment as a 100x100 grid, in the same way the environment was discretized in the MASON simulation. Each grid of the cell was modeled by a separate `PheromoneAgent`. Each `PheromoneAgent` stores the pheromone levels in the block it represents and updates them according to diffusion and evaporation at each time step. To implement the diffusion operator, we used the Euler diffusion stencil, which requires each block to know the pheromone values of its immediate neighbors. Modeling evaporation was simpler, since evaporation rates were assumed to not depend on the pheromone values of neighboring cells. At each time step T of the simulation, each `PheromoneAgent` has received a message from its neighboring agents with their pheromone levels. The agent computes its pheromone levels at time T+1 and sends the updated pheromone levels to its neighbors, priming them for the next time step. Each `PheromoneAgent` also sends updated pheromone information to all mobile agents currently subscribed to it. This can be seen in Figure 4.

Unfortunately, due to limitations in the SASSY library, we have been unable to auto-parallelize this ant simulation, but we continue to work on updating the engine and plan to add several features to facilitate this auto-parallelization process. This is discussed in Section 5.

4 RESULTS

The bytecode transformer performed very well in all tests. Our stateless ball simulation achieved a speedup of 3.44 on 4 machines (i.e., it runs 3.44 times faster than the running original sequential code (MASON only code). This is very close to the 3.52 speedup we were able to achieve with a hand-coded SASSY implementation of the same simulation. Figure 5 shows the results of running both the hand-coded and the transformed simulation on up to 4 processors (run in a distributed environment). Both of the SASSY simulations ran about 1.5% slower than MASON on a single processor, but showed significant speedup when run on several machines. For this simulation, our hand-coded version was almost exactly the same as the Soot output, but the hand-coded tests were from 1 to 2.5% faster than the Soot transformations. We believe this may be due to excessive loads and stores introduced when Soot converts from its intermediate

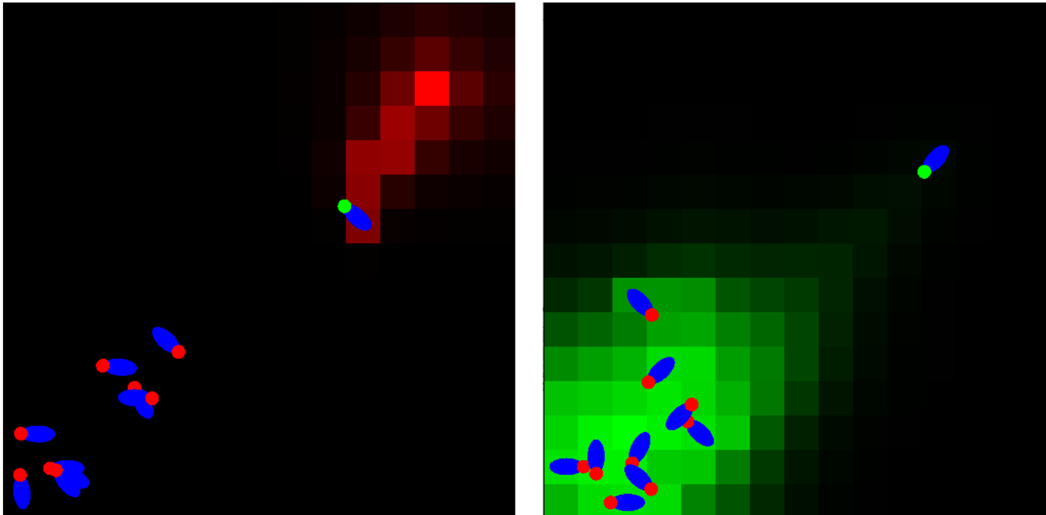

Figure 4: SASSY Ant Simulation - `findNest` Pheromone is to the left; `findFood` Pheromone is to the right.

forms back to Java bytecode (Vallée-Rai et al. 2000).

In addition to the stateless experiment, the `WorldStateAgent` has been fully implemented in SASSY and converts MASON code into SASSY. Unfortunately, SASSY does not currently support the addition and removal of agents while a simulation is running, so general MASON simulations, including the ant simulation we were testing, can not be converted until this feature is added to the SASSY kernel. This and other improvements to the system will be discussed in the next section.

5 CONCLUSIONS AND FUTURE WORK

So far, the results of our MASON-to-SASSY transformation are promising. Our implementation achieves a speedup of 3.44 on an automatically parallelized simulation running in a distributed environment of 4 machines (it runs 3.44 faster than the sequential implementation). A `WorldStateAgent` has been added to SASSY to handle all reads and writes to the state of the simulation, and we have targeted several improvements that need to be made in SASSY's architecture to better facilitate this parallelization process. As an optimistic PDES, SASSY naturally uses more memory than MASON to keep track of states in case of rollback. Since each SASSY `Agent` runs in its own thread and needs access to the world state, there is one copy of the state given to each `Agent` to prevent the need for unnecessary locks that would slow down the system. We are aware of this memory problem and have studied solutions to it in previous work (Vulov et al. 2008).

One of SASSY's much needed features is load balancing. This would speed up the performance of the simulator, and would allow for dynamic `Agent` allocation during a simulation. Without the ability to add and remove agents in the middle of execution, our approach will not be able to fully support MASON. The ant simulation we were studying relied on its ants to die and new ants to take their place throughout the life of the program. Our hand-coded simulation was unable to replicate this behavior, so we're currently looking into this feature and plan to add it to SASSY in the future.

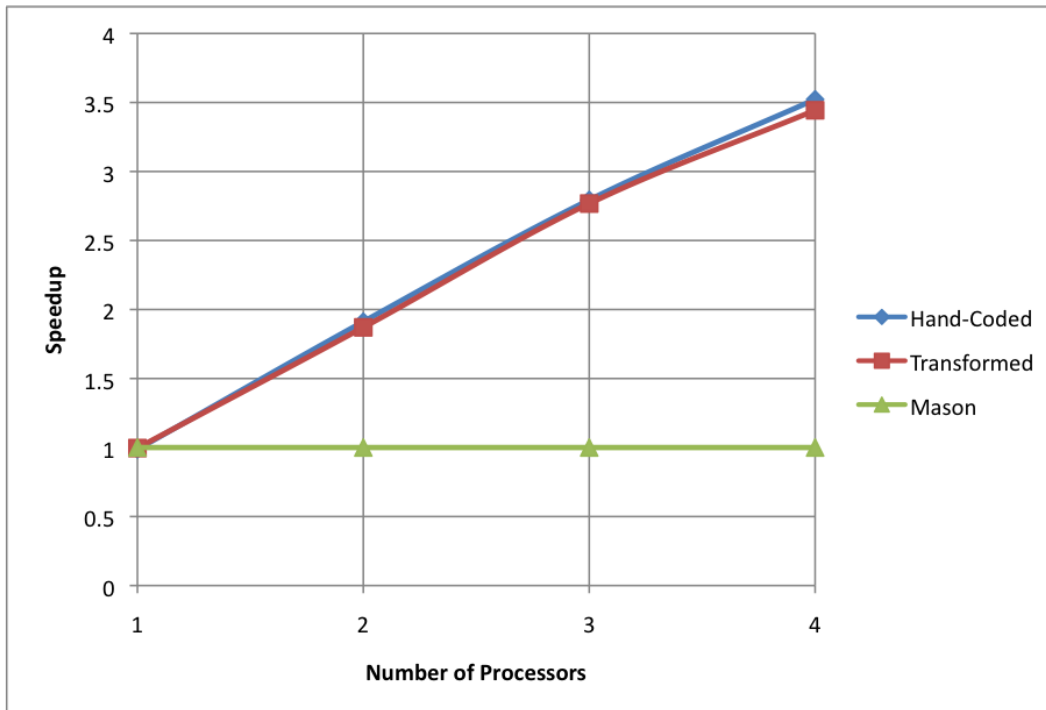

Figure 5: Number of Processors vs. Speedup (larger is better).

ACKNOWLEDGEMENTS

We would like to think George Mason University's Evolutionary Computation Lab and Center for Social Complexity for the MASON toolkit and the Sable Research Group of McGill University for the Soot optimization framework.

REFERENCES

Balch, T. 1998. *Behavioral diversity in learning robot teams*. Ph. D. thesis, Georgia Institute of Technology.

Gerkey, B., R. Vaughan, and A. Howard. 2003. "The player/stage project: Tools for multi-robot and distributed sensor systems". In *Proceedings of the 11th international conference on advanced robotics*, 317–323. Citeseer.

Hybinette, M., E. Kraemer, Y. Xiong, G. Matthews, and J. Ahmed. 2006. "SASSY: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure". In *Proceedings of the 2006 Winter Simulation Conference*, edited by L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 926–933. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Kulkarni, M., K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. 2009. "Optimistic parallelism requires abstractions". *Communications of the ACM* 52 (9): 89–97.

Logan, B., and G. Theodoropoulos. 2001. "The distributed simulation of multiagent systems". *Proceedings of the IEEE* 89 (2): 174–185.

Luke, S., C. Cioffi-Revilla, L. Panait, and K. Sullivan. 2004. "Mason: A new multi-agent simulation toolkit". In *Proceedings of the 2004 SwarmFest Workshop*, Volume 8. Citeseer.

Panait, L., and S. Luke. 2004. "Ant foraging revisited". In *Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*. Citeseer.

Vallée-Rai, R., P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. 1999. "Soot-a Java bytecode optimization framework". In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, 13. IBM Press.

Vallée-Rai, R., E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. 2000. "Optimizing Java bytecode using the Soot framework: Is it feasible?". In *Compiler Construction*, 18–34. Springer.

Vulov, G., T. He, and M. Hybinette. 2008. "Quantitative assessment of an agent-based simulation on a time warp executive". In *Proceedings of the 2008 Winter Simulation Conference*, edited by S. J. Mason, R. R. Hill, L. Moench, O. Rose, T. Jefferson, and J. W. Fowler, 1068–1076. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

COLE SHERER is a PhD student in Computer Science at the University of Georgia. He received two Bachelor's degrees, one in Mathematics and one in Computer Science, from the University of Georgia in 2007. His research interests include simulation, game programming and agent-based simulation. He competed in the ACM Collegiate Programming contests and TopCoder algorithm contests for several years. In 2009, he founded Ginger Magic Games, an Athens-based game development company. He joined the Distributed Simulation Laboratory at the University of Georgia in 2010.

GEORGE VULOV is a PhD Student in Computer Science at Georgia Institute of Technology . He received the BS degree in Mathematics and Computer Science with honors from the University of Georgia in Athens. He is a recipient of the Charles M. Strahan Award in Mathematics and the UGA Midterm Foundation Fellowship. He has competed in the ACM Collegiate Programming contests since 2005. His research interests include software engineering, computational science and agent-based simulations.

MARIA HYBINETTE is a researcher in high performance simulation systems. Her current focus is on social animal behavior modeling and financial market simulation. She is especially interested in hybrid (both micro & macro) simulation of multi-agent behavior, and mechanisms for making them faster, more effective, and more usable. In earlier work, she developed a number of methods for boosting the performance of discrete event simulations on parallel multi-processors. She lives with her husband Tucker, 3 children, 2 cats, 30,000 bees (outside) and a colony of ants (inside). When she's not teaching or researching, she loves learning about photography, reading and playing with her family.