Published in final edited form as:
ACM Trans Model Comput Simul. 2011 December ; 22(1): 2–. doi:10.1145/2043635.2043637.

A Distributed Platform for Global-Scale Agent-Based Models of Disease Transmission

Jon Parker and
The Johns Hopkins University

Georgetown University

Joshua M. Epstein
The Johns Hopkins University

Abstract

The Global-Scale Agent Model (GSAM) is presented. The GSAM is a high-performance distributed platform for agent-based epidemic modeling capable of simulating a disease outbreak in a population of several billion agents. It is unprecedented in its scale, its speed, and its use of Java. Solutions to multiple challenges inherent in distributing massive agent-based models are presented. Communication, synchronization, and memory usage are among the topics covered in detail. The memory usage discussion is Java specific. However, the communication and synchronization discussions apply broadly. We provide benchmarks illustrating the GSAM's speed and scalability.

General Terms

Design; Performance

Additional Key Words and Phrases

Epidemiology; agent-based modeling; agent behavior

1. INTRODUCTION

Modern mathematical epidemiology began with the Kermack-McKendrick model of 1927. This elegant system of differential equations posits perfect mixing in the population, with individuals moving from the susceptible pool to the infected one to the removed (recovered or dead) one. Within these pools, or "compartments," there is no diversity among people, and no adaptation in their behavior. Despite these strong idealizing assumptions, the approach fundamentally illuminated the threshold ("tipping point") nature of epidemics and explained "herd immunity," wherein immunity of a *sub*population can make outbreaks

Author's address: J. Parker, Department of Emergency Medicine and The Center for Advanced Modeling in the Social, Behavioral, and Health Sciences, The Johns Hopkins University, Baltimore, MD 21209; jparker5@jhmi.edu.

fizzle, protecting the entire herd. This remains the goal of vaccination strategy to this day, and the original equations spawned a voluminous and valuable scientific literature [Anderson and May 1993; Keeling and Rohani 2007].

Despite their fundamental contributions, differential equation models are ill-suited to representing complex modern social networks and contacts between distinct individuals, interacting directly with one another (not in a well-mixed population) as they move about spatially, adapting their behaviors–perhaps irrationally–based on disease prevalence. With the advent of modern computing, another tradition has emerged which embraces this complexity: the agent-based computational model. This approach has made deep inroads into infectious disease modeling, in some respects displacing differential equations. The Global-Scale Agent Model (GSAM) represents the state-of-the-art in agent-based epidemic modeling.

1.1. Defining Feature of Agent-Based Models

Simply put, agent-based models (ABMs) are artificial societies [Epstein and Axtell 1996]. In an orthodox ABM, every single individual is represented as a distinct software object. There is no aggregation into homogeneous pools. Agents are heterogeneous and can differ in myriad ways (e.g., age, disease state, behavioral rules). Events unfold on an explicit space of some sort – a social network, a city, or a multinational region, for example. Within this interaction space, agents act autonomously, executing some itinerary, such as going to school or work. Typically, these cyber-people have limited (often local) information, and limited cognitive capacity. They are boundedly rational, to use Simon's [1982] phrase, and their behavior may adapt depending on their perceived situation [Epstein et al. 2008]. Finally, and crucially, agents interact directly with one another. So, in a strictly agent-based infectious disease model, a dendrogram showing exact chains of agent-to-agent transmission is constructible [Epstein 2007]. Even in stochastic models, for any particular realization, the precise sequence of person-to-person transmissions can be reconstructed and displayed. For a computational model to qualify as an ABM in this most catholic sense, it must pass this "dendrogram test."

1.2. Applications of Agent-Based Infectious Disease Models

Agent-based infectious disease models (also termed *individual-based models*) are a comparatively recent child of computing. Epstein and Axtell [1996] published such a model in 1996. Since then, model scales have grown steadily, from county-level [Burke et al. 2006; Epstein et al. 2004], to city level [Eubank et al. 2004], to the small nation [Ferguson et al. 2005; Longini et al. 2005], to US scale. [Ferguson et al. 2005]. Cooley et al. [2008] examine a number of the more prominent published models.

These models have been highly influential, and have been used by governments and international health organizations to devise containment strategies for smallpox, MRSA, dengue, hoof and mouth disease, Avian Flu, and now H1N1 swine flu. The National Institutes of Health MIDAS (Models of Infectious Disease Agent Study) Project, the Johns Hopkins Medical School's DHS-funded PACER (Preparedness and Catastrophic Event Response) Project and the CDC-University of Pittsburgh Public Health Adaptive Systems (PHASYS) are prominent US examples. The Global Scale Agent Model presented here is employed in all these efforts.

1.3. Three Distinctive Features of the GSAM

The GSAM is unprecedented in its scale, in its speed, and in being Java based. Regarding scale, there are global models, but they are not agent-based. There are agent-based models, but they are not global. The GSAM is the first planetary scale strictly agent-based model.

Indeed, no other ABM of epidemics includes more than a few hundred million individuals, where–as we demonstrate–the GSAM can efficiently run with over 6 billion individuals simultaneously in memory. Regarding speed, the GSAM's US submodel (300 million agents) executes in roughly twenty minutes, where the national scale competitors require many hours on similar computers. The GSAM is dramatically faster than these, despite their presumed advantage of being written in C or C++. GSAM does this while remaining portable because it is written entirely in Java. We believe the GSAM is one of few (if any) distributed agent-based models that does not suffer major performance penalties due to internode communication. This all argues for a fundamental reevaluation of Java's place in high performance computing.

Precisely how this was accomplished is the subject of this article. Before presenting implementation details, however, it is important to preempt a possible confusion.

1.4. Calibration to Particular Disease Irrelevant

The GSAM is a platform. It is not a model of any particular disease. The present exposition of the general platform obviously requires us to specify a contagious disease of some sort. To this end, we will use a plausible representation of the H1N1 (Swine flu) virus, employing assumptions consistent with current scientific publications in this area. But we explicitly renounce any claim to have "validated" the H1N1 model proper, in so far as this term denotes calibration against data. This is simply not our aim. To criticize the platform based on objections to illustrative H1N1 disease specifics misses the point.

2. FLEXIBILITY AND LANGUAGE CHOICE

It would be shortsighted to build a powerful modeling platform that supports only a single disease and a single hardware environment. With this in mind, maintaining flexibility was the chief high-level design goal of development. In particular, we wanted the GSAM to be portable across operating systems and hardware configurations. We also needed the GSAM to support multiple diseases (perhaps circulating simultaneously). This disease flexibility requires the GSAM to support a wide array of possible individual behaviors. For example, the behavior important to an influenza simulation (school contacts) differs radically from the behavior important to an HIV/AIDS simulation (sexual contacts, needle-sharing).

Due to the hardware and software flexibility requirements it is not surprising that the GSAM is implemented in Java. Java is a modern programming language that supports object-oriented (OO) design. A proper OO design enables the GSAM to easily exhibit the behavior flexibility we required by extending a class here and implementing an interface there. Using Java also allows the GSAM to be quickly shuttled from one operating system to the next. This Java implementation is also efficient on both shared-memory and distributed-memory hardware. In short, Java enables the GSAM to easily meet its flexibility goals.

Java has other nontrivial benefits. As a widespread language, there are many mature, fully documented utilities available to leverage. These include the data structures of java.util, the communication utilities of java.rmi, the technical computing resources of the Colt Project, and the graphics utilities of JFreeChart. The multithreaded programming resources in java.util.concurrent (concurrent data-structures, semaphores, thread pools, and executors) are invaluable when creating multithreaded programs. Java does have one noteworthy constraint – remote method invocation (RMI) has large latency. However, we show how this problem can be easily circumvented.

3. A PARALLEL APPROACH, DESIGN SYNOPSIS

There are two simple facts that must be dealt with when developing a large-scale agent-based model. First and foremost, a capable design must have access to large swaths of memory. There is no evading the fact that storing 6 billion agents and their itineraries will take up plenty of memory. The second unavoidable fact is that a singlethreaded design is unreasonable for large populations. It will simply take too long to return results. It will also fail to capitalize on the additional computing resources available on today's multicore and multiprocessor machines.

Together, these facts compel an effective model to be parallel. Accordingly, the GSAM is a highly parallel distributed model. The GSAM's workload is distributed across two layers. It is first distributed across a node layer – a node is a Java Virtual Machine (JVM) dedicated to running the GSAM. Each node then distributes its workload among its working threads. This dual-stage distribution has two main advantages. First among them is that it allows the GSAM to support different arrangements of computing resources. For instance, a GSAM run can be executed entirely in one node that supports *n* working threads. Or the identical population can be simulated across *n* nodes, where each node operates a single thread. Appendix A contains a graphic that depicts three equivalent modeling arrangements. This flexibility enables a GSAM to easily run on different hardware platforms.

The second benefit of a dual-stage distribution scheme is that it provides an excellent avenue to study many variables that affect model performance. For instance, since the nodes can be remote or local, we can measure the performance of communication between two nodes on the same computer vs. two nodes on different computers. Or we can compare communication between two threads of the same node vs. communication between two threads from different nodes. Performing these comparisons allows the performance of the model to be better understood, and facilitates the creation of a faster, more efficient model.

3.1. Preparing for Distribution

Before the GSAM can distribute its population across its computing resources, it must partition its population into manageable pieces. Throughout this article, we refer to these manageable pieces as ModelBlocks (MB). Each ModelBlock represents a group of agents who live close to one another in geographic space. MBs could represent all the agents who live in a city block, a zip code, a census tract, or perhaps a square kilometer. The size and shape of ModelBlocks are likely to depend strongly on the format of the input data. For instance, given access to US Census Tract data it may be convenient to create a GSAM model in which each US Census Tract is represented by exactly one ModelBlock, while a modeler who has access to the grid-based LandScan dataset may choose to aggregate many adjacent grid squares into one ModelBlock (The LandScan dataset is a 43200 by 20880 grid of estimated populations).[1]

Once the population has been partitioned into MBs, the GSAM is ready to distribute the population (i.e., the ModelBlocks) across its computing resources. As with many distributed computing problems, the best way to allocate the ModelBlocks across the available resources is not easily determined. For example, a GSAM simulation of an epidemic within the United States performs best when each computing resource is given a contiguous region to simulate. On the other hand, a GSAM simulation of a global epidemic performs best when geographic regions are divided between as many computing resources as possible.

[1] http://www.ornl.gov/sci/landscan/

This difference in peak performance is caused by differences in ModelBlock size. A global simulation uses larger ModelBlocks (in both population and area). Therefore, agents in "global size" ModelBlocks are more likely to interact with agents from within their own ModelBlock. This results in fewer messages between MBs (on a per-capita basis). The reduced communication burden allows an optimal distribution strategy to focus closely on balancing the node's estimated workloads. However, when more messages must be sent between ModelBlocks, as in the US example, an optimal distribution strategy will reduce the substantial cost of sending and receiving these messages by grouping ModelBlocks that interact with each other onto the same node.

We strongly suspect that no single distribution strategy will be best for all possible GSAM models. If no universally best distribution strategy exists, then any single GSAM model will only attain its maximum possible performance using a customized distribution strategy. Fortunately, a time consuming search for this customized strategy is not necessary because adequate performance can be achieved using round-robin, or purely random, allocation schemes (see Section 6).

Finally, dynamic load balancing [Carothers and Fujimoto 2000; Deelman and Szymanski 1998; Peschlow et al. 2007] is worth a brief discussion. The GSAM was originally designed to support dynamic load balancing by transferring ModelBlocks from one node to another. However, the extremely high cost of removing a ModelBlock (along with its agents and their events) from a node and inserting it into another node renders dynamic load balancing a clearly losing strategy. Therefore, the GSAM does not implement any dynamic load balancing scheme. Fortunately, significantly unbalanced nodes can usually be avoided when ModelBlocks are allocated to nodes in a round-robin style. Randomly allocating ModelBlocks to nodes also prevents dramatic load imbalances.

3.2. Simulation Overview

One reason the GSAM platform is fast is that it simulates only active agents although all agents are in memory. Who are these active agents and why is this an allowable reduction in scope? Consider this analogy: a row of 6 billion contiguous dominoes is given. Some domino (the index case) is toppled, and a contagion of falling dominoes ensues. You wish to simulate the progress of this wave. It would be absurd to loop through the entire list of all 6 billion dominoes at every simulated time-step. Each trip through the list would examine billions of dominoes when only a handful are changing state at any one time. It is far more efficient to maintain a list of active (i.e., falling) dominoes and operate only on this set. Proper implementation of this active set modeling scheme requires that each domino correctly determine which dominoes it will effect when active. This allows active dominoes to promote other dominoes to the active set at the appropriate time. Predictably, when a domino completes its fall it will remove itself from the active set.

From the domino example we see that, to track a contagion, simulating everyone's entire day-to-day schedule may not be necessary. But, for the GSAM to use this active set approach its active agents must be able to determine who they will interact with while active. Fortunately, this is easy to do in an agent-based model.

The agent-to-agent contacts that drive disease transmission can be split into two broad categories: repeat contacts and random contacts. Both of these contact types can be faithfully recreated when just the active agent is known. Repeat contacts are easy to correctly implement because agents know whom they interact with on a regular basis. Examples of repeat contacts are intrafamily contacts, contacts between friends, and contacts between coworkers or classmates. Random contacts can also be implemented appropriately. Active agents can select a properly distributed random agent from the entire population

using a special probability mass function. The active agent can then interact with this randomly selected agent, thus completing a random contact, see Section 3.4,"Random Agent Selection and the Contact Matrix," for greater detail.

Together, explicitly stored social networks and random agent selection enable realistic simulation of both repeat and random contacts. This eliminates the need to explicitly simulate every single agent's day-to-day activities and allows the GSAM to simulate only the behavior of active agents. Agents are considered active when they are contagious (i.e., infectious) or symptomatic or both. The behavior of contagious agents is obviously crucial to an epidemic's progression. However, the relevance of merely symptomatic agents requires an explanation. Symptomatic agents must be modeled explicitly because their behavior while symptomatic (but not yet contagious)may alter the simulation. For instance, when an agent showing symptoms opts to go to the hospital and receive treatment he may be less likely to infect other agents when he does become contagious or perhaps his treatment will prevent infectiousness entirely.

Active agents are simulated using detailed itineraries that are randomly generated on-the-fly (i.e., during a run). These itineraries are generated by a personalized AgentEvent object that is created when an agent is promoted to the *active set*. Itineraries explicitly model three different event types (behaviors). Those event types (behaviors) are family contacts, coworker/classmate contacts, and random contacts. Incidentally, several other epidemic models also use these social networks (family, work/school, random) [Ferguson et al. 2005, 2006; Glass et al. 2006; Longini et al. 2004]. For more information about agent itineraries see "Agent Itinerary Generation" below.

These AgentEvents are stored in a thread's only priority queue. This priority queue contains all the events that take place within any of the ModelBlocks a thread has been allocated. When a GSAM thread is running it polls (removes) the event with minimum time, typically an AgentEvent, from its sole priority queue and implements the appropriate event. Execution proceeds in this manner until the simulation is complete.

Now that we have a "forest" level view of the GSAM we can take a more in-depth "tree" level view of certain design aspects. There are a few GSAM design features that warrant very detailed discussion. Among them are synchronization/communication and memory usage. We will conclude with a detailed discussion of model performance and scaling.

3.3. Agent Itinerary Generation and Behavior

First, an epidemic model needs to know when people get sick, whom sick people contact, when those contacts occur, and when the sick recover. An agent's itinerary is a sequence of events that answers these (and possibly more) questions. With so much riding on an agent's itinerary it is vital that these itineraries are generated in a flexible, sensible way.

The GSAM platform needs the itineraries to be flexible so that it can support different activities for different people. For instance, the itinerary of a stay-at-home parent should include many midday family contacts while the itinerary of a parent who works a standard 9-to-5 job should virtually exclude family contacts during business hours. The GSAM platform is built specifically to enable differences like this to be easily incorporated into agent itineraries.

Currently, an agent's itinerary includes four different classes of events: family contacts, coworker/classmate contacts, random contacts, and disease updates. The three contact classes are generated using three different easily customizable behavior streams. A behavior stream is nothing more than a graph of occurrence probability for the relevant event versus

the event's occurrence time. The resulting graph defines the instantaneous rate (intensity) function $\lambda(t)$ at each simulation time t for a nonhomogenous Poisson process (NHPP) (for some, the time variable may not be discretized finely enough to warrant distinction as a Poisson process). This NHPP is used to generate the times at which contacts will occur. Figure 1 illustrates three behavior streams that could generate a plausible itinerary for an agent during a normal business day.

It is easy to support many different types of agents by defining many different behavior streams. It is also easy to support dramatic shifts in behavior by abruptly switching the behavior streams from which an agent is generating its itinerary. Swapping behavior streams makes it easy to implement weekend behavior that is vastly different from weekday behavior. It is also useful when modeling how an agent's itinerary changes after he begins to exhibit symptoms of a pathogen.

The flexibility of the behavior stream approach also makes modeling different diseases more manageable. This is because the interpretation of a contact may change when the disease being modeled changes. For instance, a contact itinerary suitable for a flu model that includes many agent-to-agent contacts per day is clearly inappropriate in a model of HIV/AIDs. The ease with which the behavior streams can be altered makes authoring a new GSAM less onerous.

Agent itineraries also include disease update events. These events move an agent through the natural history of disease and are generated in accordance with the biology of the pathogen being modeled. Currently, an agent's disease status falls into one of the following states: Dead, Recovered, Susceptible, Noncontagious-Asymptomatic, *Noncontagious-Symptomatic*, *Contagious-Asymptomatic*, and *Contagious-Symptomatic*. If an agent's disease status is one of the italic states he is considered active, and his itinerary will be generated and modeled. Notice, only three of the four infected states are modeled.

If desired, the list of possible disease states could be expanded. For instance, smallpox comes in four different varieties (ordinary, modified, malignant, and hemorrhagic) and it may prove useful to include this distinction in the list of disease states. (However, other programming solutions exist, so expanding the state list is not strictly necessary.)

For more information on the AgentEvent class (the class that generates the itinerary) see the Section 4.3, "The Priority Queue and Its Contents". This section describes the exact implementation in greater detail.

3.3.1. The Importance of Behavior—An acknowledged weakness of traditional epidemic modeling is that behavioral adaptation is ignored [Epstein et al. 2008; Kremer 1996]. The GSAM is useful to decision-makers because it is easy to implement itinerary changes induced by policies such as school closures and travel restrictions. It can also incorporate endogenous behavioral changes due to fear or prevalence-elasticity during a pandemic. The GSAM's speed offers rapid feedback on the potential impact of various policies, as well as behaviors such as vaccine refusal or nonadherence to health recommendations.

3.3.2. Computing Next Event Timing—In many simulations computing the occurrence time of a NHPP's next event is the single most important determinant of overall simulation performance. Hence, it is important to use an efficient computation method [Lee et al. 1991]. Space constraints prohibit a full treatment here. However, much more information about this topic can be found in the online supplement.

3.4. Random Agent Selection and the Contact Matrix

In order to justify simulating active agents exclusively, the GSAM platform must be able to faithfully recreate random contacts when only the active agent is known. This means the GSAM platform must have a reasonable way to select a random agent to pair with an active agent. Making this selection requires two decisions. The first decision is determining in which MB the randomly contacted agent lives. The next decision is determining who in that MB will be the random contact. Correctly making these decisions are the key components to creating a model with realistic random contact dynamics.

For now, assume the GSAM is given a matrix P where matrix entry P_{ij} is the probability that a person from MB i travels to MB j. This matrix is a trip distribution matrix (given that a trip occurs). Treating P as input allows the GSAM to separate travel behavior assumptions from the rest of the platform.

Using P, the GSAM computes another matrix C dubbed the *contact matrix*. The entry C_{ij} is the total probability that an agent from MB i contacts an agent from MB j. Specifically, these entries give the probability that an agent from MB i contacts an agent from MB j in a mutual destination MB k, summed over all$k$. Pseudocode that generates C is shown here (this pseudocode makes use of MATLAB function names).

```
POPS = ModelBlock populations //a row vector
TRAVELERS=diag(POPS)*P //a matrix
TOTALS=POPS*P //a row vector of column sums
INTERACT_ij = TRAVELERS_ij/ TOTALS_j //a matrix
C=P*(INTERACT) //the contact matrix
```

Once C has been computed, selecting a random contact is straightforward. The GSAM knows where the active agent is from. So, suppose she is from ModelBlock i. The GSAM must convert the ith row of C into a probability mass function (PMF). The GSAM platform uses this PMF to select a random ModelBlock M. The randomly contacted agent will be selected from the population of ModelBlock M.

The randomly contacted agent should not be selected from the population of M uniformly. The selection probability must be weighted according to the likelihood that an individual agent would be a part of a random contact (as defined by his behavior stream). For instance, if an agent's behavior stream dictates that he has zero probability of making a random contact at time t, it would be improper to pair this agent with another at time t. The GSAM platform adjusts for differences between agents' behavior streams using a rejection sampling technique [Robert and Casella 2004].

It is important to note that, due to memory requirements, the contact matrix places a soft upper bound on the number of ModelBlocks that can plausibly be simulated. For example, storing a 15,000 × 15,000 matrix of doubles requires 1.8GB of memory while a 20,000 × 20,000 matrix requires 3.2GB of memory [Gosling et al. 2005]. Incidentally, there are approximately 31,000 zip codes in the continental US. Storing a matrix of this size (almost a trillion entries) requires 7.67GB of memory.

3.4.1. The Travel Matrix P—The prior section treats the matrix P as given to emphasize the fact that the GSAM platform always uses P the same way regardless of how P was created. Obviously, the method used to arrive at P should be justifiable. Currently, the GSAM computes P by first computing a preliminary matrix F. The F matrix contains the

estimated flow (i.e., number of trips) between different ModelBlocks. For example, if the entry F_{ij} = 20, then it is assumed that 20 people start trips in MB *i* and end their trips upon arrival in MB *j*.

F is computed as follows:

$$F_{ij}=\frac{(pop_i \cdot pop_j)^a}{cost_{ij}^b}$$

Here $cost_{ij}$ is an estimate of the travel cost for trips between ModelBlocks *i* and *j*, the *pops* are the populations of the respective ModelBlocks, and *a* and *b* are parameters.

At this point, the GSAM can compute the *P* matrix by dividing each entry in *F* by its corresponding row sum as shown here:

$$P_{ij}=\frac{F_{ij}}{\sum_i F_{ij}}$$

The *F* matrix is an example of a gravity model (so-called due to its algebraic resemblance to Newton's Law of Gravitation, with the *pops* interpreted as mutuallyattracting masses, etc). In addition to modeling travel/traffic patterns, gravity models have been used to model trade flows and migration [Bergstrand 1985; Isard 1954; Voorhees 1956].

4. COMMUNICATION AND SYNCHRONIZATION

4.1. Design

Without doubt, the careful design and implementation of internode and interthread communication is the most important determinant of the GSAM's performance. Some of the following discussion will be Java specific. However, the design aspects most crucial to performance could likely be incorporated into models/platforms written in different programming languages.

While a thread is repeatedly processing events in its priority queue, it will occasionally encounter an event that requires interaction between agents assigned to different threads. Since this event is expensive to implement instantaneously, it will set aside this particular event for later execution. A record of this unexecuted event will be saved in a new OffThreadContactEvent (OTCE) object. After this record is created, the executing thread will continue processing events within its priority queue. Each OTCE will contain enough information to enable the receiving thread to implement the previously scheduled, but unexecuted interaction. Eventually, these OTCEs must be sent to the other threads for execution.

At first glance, the process of saving some events for later execution appears severely flawed. Here is an example of the apparent problem. Let OTCE α represent a contact between two agents that should occur at time 172. Yet, α gets transmitted and executed at time 200. How has this not corrupted the timeline of the model?

The answer lies in the incubation and latent periods of the disease (see Figure 2). If an agent is exposed and infected with a disease at time *x* but does not alter his behavior or infect other agents until time (*x* + *y*) then the newly infected agent is effectively unchanged until time (*x* + *y*). Therefore, notice of his infection can be delayed as far as, but not including, time (*x* +

y). Thus, the maximum allowable time between communication events *y* is strictly dictated by the minimum of the incubation and latent periods of the simulated disease.

Using this rule to set the communication interval guarantees that communication occurs frequently enough to ensure that agents who are infected by OTCEs (that are always delayed by some amount) can have their "move to the active set" time assigned in such a way that they always enter the active set at the correct time despite the fact that the notification of infection arrived late. In other words, the communication delay had no effect.

Curiously, delaying communication enables some agents to be infected twice – loosely speaking. Consider an OTCE that is guaranteed to infect agent *i*. If this OTCE is created at time 172 and sent at time 200 it is still possible for agent *i* to be infected in the interim, say at time 184. In this case, agent *i* must have his infection time rolled back from 184 to 172, when his infection first occurred. These simple rollbacks are the only corrections that delayed communicate necessitate. The third portion of Figure 4 depicts this minimal rollback.

Since communication can be delayed a small amount without consequence, periodic bulk communication is best. It is easy to code and debug. It has the least overhead. It provides a convenient place to collect data. And it facilitates reproducibility. This is how the GSAM platform manages communication between all threads. A flowchart of the communication is shown in Figure 3.

4.1.1. Another Interpretation—The GSAM platform could be considered a platform for Parallel Discrete Event Simulation (PDSE). Using the terminology of PDSE, a GSAM is an optimistic simulation (i.e., some events can be executed out of time sequence) [Fujimoto 1990; Perumalla 2006]. Since the GSAM platform is optimistic it does not suffer from the severe drawbacks associated with coordinating conservative simulations. Nor does it suffer from the main drawback of optimistic modeling – correcting out-of-order computations. Typically, optimistic simulations must be prepared to roll back events that were executed too soon. Fortunately, the GSAM does not need to create expensive checkpoints or waste substantial execution-time correcting prior calculations, because its rollback mechanism is trivial. Also, the rule for setting the communication interval prevents the possibility of a cascade of rollbacks. In short, the GSAM platform avoids the biggest problems of both optimistic and conservative simulation.

4.2. Implementation

Periodic bulk communication does not immediately confer excellent model performance. However, periodic bulk communication allows for three important optimizations that otherwise would not be possible. One optimization involves eliminating the threat of data corruption due to concurrent manipulation. The other two optimizations reduce communication overhead by orders of magnitude [Mathew and Roulo 2003]. These reductions in communication overhead are the most important optimizations throughout the entire GSAM platform.

Recall that each ModelBlock (and its data) is assigned to a specific thread. These assignments form a partition of the population that is used to define each thread's exclusive domain. Normally, threads in a multithreaded environment spend a nontrivial amount of time acquiring and releasing semaphores and/or object locks that prevent data corruption due to concurrent access [Christopher and Thiruvathukal 2001]. If threads are restricted from operating outside their exclusive domains, then those threads can operate without the usual performance-degrading object-locking schemes. Not only does this improve performance, but it also makes programming notably easier.

A proper discussion of the next two optimizations requires a little Java-specific knowledge. Java's Remote Method Invocation (RMI) makes implementing inter JVM (e.g., inter node) communication extremely simple. Communication becomes simple because RMI leverages other expensive powerful operations to do its work [Java 6 SE RMI documentation]. Namely, RMI automates the serialization (deconstruction of an object into a stream of bits), transmission, and deserialization (the reconstruction of the original object from the bit stream) of every object it transmits between JVMs. These three automated steps are clearly nontrivial, and that is reflected in the execution cost of RMI calls. While RMI makes coding simple, using it inefficiently can severely hinder performance.

When communicating in bulk the entire queue of unexecuted OTCEs can be sent using one RMI call. This single bulk message can eliminate literally millions of expensive RMI calls per communication period. This optimization is so powerful that going without it may increase computation time by a factor of 100 or more.

The second possible overhead reduction is easy to overlook because RMI makes implementing its alternative so graceful. It is natural to code a communication scheme like this.

```
/* Send the raw OTCE array to another node using RMI. */
public void exportEvents(Node nd, OffThreadContactEvent[] events) {
try {
nd.importEvents(events); //the RMI call
{ catch (RemoteException re) {
//handle an error here
}
}
```

In fact, this is how communication was originally implemented in the GSAM. This communication scheme does reduce the number of RMI calls (which is the most crucial optimization), but it fails to reduce the number of serialization/deserialization pairs.

It is more efficient to transmit a single large bucket of primitive arrays (i.e., double[], boolean[], etc.) rather than an array of many small objects. When RMI serializes an array of objects it must serialize each and every object within that array [Java Serialization Documentation]. If, however, that array of objects is transformed into multiple arrays of primitives, then Java can quickly send those primitive arrays over RMI. A code snippet that implements this optimization is shown here.

```
/* Build a “bucket object” to send, do not send OTCE array. */
public void exportEvents(Node nd, OffThreadContactEvent[] events) {
OTCEBucket bucket = buildBucket(events);
try {
nd.importEvents(bucket);); //the RMI call
{ catch (RemoteException re) {
//handle an error here
}
}
```

Table I demonstrates the enormous power of these two optimizations. This table shows how long it takes to transmit 10 million numbers between two JVMs using RMI calls with different signatures. The first two rows in this table use RMI calls that only send one piece of data per call. The last two rows use RMI calls that bundle all the data into one array before making a single RMI call. The 1st and 3rd rows leave the original data in objects and require significant serialization/deserialization. The 2nd and 4th rows transmit only primitive data and require no serialization. Even though this example does not use OTCEs, it is clear that assembling and transmitting a bucket of primitive arrays is vastly preferable to other communication strategies.

4.3. The Priority Queue and Its Contents

While a GSAM thread is operating, it is constantly finding, removing, and executing the SimulationEvent with the minimum time (SimulationEvents will be discussed in depth momentarily). Typically, executing one SimulationEvent produces another SimulationEvent that must be added to the collection of SimulationEvents. Occasionally, a GSAM thread must remove an arbitrary SimulationEvent from its collection of SimulationEvents.

Given this description, it seems likely that the GSAM would use java.util. PriorityQueue (or its Colt Project equivalent) to store its SimulationEvents. This class supports both insertion of new items and removal of the minimum item in $O(\log(n))$ time. But, this class cannot efficiently remove an arbitrary item because it is implemented using a heap. Consequently, the GSAM attains better performance storing its collection of SimulationEvents in a binary tree.

It may be a slight misnomer to use the term priority queue throughout this article when, in fact, the GSAM uses TreeMaps, which are red-black trees. However, since the GSAM makes such heavy use of the *put* and *pollFirstEntry* operations it makes sense to use the term priority queue because that is the spirit in which the TreeMaps are used.

As noted, priority queues (TreeMaps) contain SimulationEvents. SimulationEvent is an abstract class that implements the Comparable interface. Exactly four classes extend SimulationEvent. This shared base class allows all subclasses to share a single sorting methodology. A code skeleton for the SimulationEvent class is as follows.

```
public abstract class SimulationEvent implements Comparable {\{}
/** When this SimulationEvent should be implemented. */
protected int time;
/** The ModelBlock associated with this event. */
protected ModelBlock place;
/** A tiebreaker for use when sorting SimulationEvents. */
protected final int tiebreaker;
/** Override this method to define different events. */
public abstract implementEvent();
/** Sort SimulationEvents by time, then place, then tiebreaker. */
public int compareTo(Object otherEvent) {
//implementation not shown to save space
}
}
```

ReconcileEvent (RE) is the simplest SimulationEvent subclass. Exactly one RE is placed in each priority queue. When this RE reaches the top of the priority queue, it forces the

executing thread to go idle. A thread's final acts before going idle are to increment the ReconcileEvent's time variable, insert the RE back into the priority queue, and inform its parent node that it has gone idle.

Once a thread is idle, it is ready to send and receive OTCEs. Communication of OTCEs between threads occurs when all threads throughout the GSAM have gone idle. This ensures that OTCEs are sent and received in a predictable order. It is important to note that a thread's current time goes backwards when OTCEs are imported.

LogUpdateEvent (LUE) is the third subclass of SimulationEvent. LogUpdateEvents prompt periodic data collection. When a ModelBlock is first allocated to a thread, an LUE dedicated solely to that ModelBlock is put in the priority queue. When a LogUpdateEvent is removed from the top of a priority queue it prompts the executing thread to collect and record data about its dedicated ModelBlock. The LUE has its time variable incremented by a fixed interval and is then inserted back into the priority queue. Eventually, each ModelBlock's LUE reappears at the top of the priority queue. In this manner, LogUpdateEvents ensure that all relevant data is regularly collected over the duration of a GSAM simulation. When a simulation is complete, these logs are assembled into a single master log that contains all collected data.

The last subclass of SimulationEvent is AgentEvent. AgentEvents are created when agents are promoted to the active set due to their impending symptoms/ contagiousness. AgentEvents are still sorted according to the time variable they inherit from SimulationEvent even though the AgentEvent class declares four new time variables. The AgentEvent class ensures that the time variable from Simulation-Event is always set to the minimum of familyEventTime, randomEventTime, cowork-erEventTime, and diseaseEventTime. When an AgentEvent object is removed from the top of the queue, it may prompt the execution of one of four different subevents: family contact events, random contact events, coworker/school contact events, or disease progress events. After the appropriate subevent is executed, the corresponding time variable is recomputed (using behavior streams described in Section 3.3, "Agent Itinerary Generation and Behavior"). The time variable inherited from Simulation-Event is then recomputed, and the AgentEvent is reinserted in the priority queue. The first and last subevents an AgentEvent will call for are both disease progress events. The first disease progress event occurs when the agent becomes symptomatic or contagious, whichever happens first. The last disease progress event occurs when the agent recovers (or dies) from the disease. A code skeleton of AgentEvent is shown here.

```
public class AgentEvent implements Comparable {
/** The ID_Number of the person this scheduler tracks. */
protected int personIndex;
/** When this agent next contacts a family member. */
protected int familyEventTime;
/** When this agent next contacts a random person. */
protected int randomEventTime;
/** When this agent next contacts a classmate/coworker. */
protected int coworkerEventTime;
/** When this agent shifts from one disease state to another */
protected int diseaseEventTime;
/** Override this method to define different events. */
public abstract implementEvent();
```

```
/** Sort SimulationEvents by time, then place, then tiebreaker. */
public int compareTo(Object otherEvent) {
//implementation not shown here to save space
}}
```

As evident from the AgentEvent code skeleton, AgentEvents can be considered a bundle of four different events. We do not create separate FamilyEvent, RandomEvent, CoworkerEvent, and DiseaseEvent classes because an implementation with separate classes wastes memory. If these classes were created, then objects of those types would need to be instantiated. Then, once constructed, each of these objects would necessitate another Map.Entry object that would otherwise be unnecessary. Bundling these events into a single class also reduces the total number of events in the priority queue by almost 75% (we can assume that the number of LUEs and OTCEs will be small compared to the number of AgentEvents) (see Figure 4). Section 6 of this article shows why this 75% reduction in object count is important in terms of memory.

4.4. Random Numbers and Reproducibility

It is vital for output from GSAM models to be reproducible. Reproducibility enables the creation of more relevant performance benchmarks. Reproducibility helps developers track down bugs. And most importantly, reproducibility allows results to be independently verified. Please note “reproducibility” in this context means only “generate the same result” and does not require identical low-level memory behavior. Obviously, since Java does not offer direct memory management the latter requirement would be inappropriate.

In general, a simulation using random numbers must meet three criteria for its results to be reproducible. It must use a stream of random numbers that can be recreated exactly. It must use those numbers in the identical order. Finally, all modeled behavior needs to be deterministic or draw random numbers from the proper stream. If any of these criteria is not met, the results from a simulation cannot be reliably recreated.

The GSAM uses the Mersenne Twister-based random number generator in the previously mentioned Colt Project. This class can produce a reproducible stream of pseudorandom numbers provided the RNG is initialized with the same seed before any random numbers are drawn. Alternative generators are easily substituted if desired. Demonstrating that GSAM models meet the second criteria requires detailed knowledge of the inner workings of the GSAM. The crucial aspect is the method by which SimulationEvent objects are ordered in a thread’s priority queue. Notice, the SimulationEvent code skeleton above declares two variables: *time* and *tiebreaker* (three variables are declared, but only two are relevant here). The priority queue obviously implements events with earlier times first. But, due to the coarse-grained time scale (1 time step is about 15 minutes) and the large number of agents that are allocated to each thread, there will be many SimulationEvents that have equivalent *time* values. These ties must be broken in a systematic way so that the behavior of the priority queue (which sorts the Simulation event by their compareTo method) is well defined. Not surprisingly, the *tiebreaker* variable is used exclusively to break these ties.

Each GSAM thread maintains a counter whose sole purpose is to sequentially number the SimulationEvents. When the SimulationEvent constructor is called, it is passed a reference to a ModelBlock. This ModelBlock maintains a reference to the thread to which it was allocated. This allows every new SimulationEvent to trace back from ModelBlock to the corresponding thread and finally to this counter. The new SimulationEvent sets its *tiebreaker* variable to the counter’s current value after which the counter is incremented. Once set, the *tiebreaker* variable never changes. This guarantees that all ties will be broken

in a way that is consistent from run to run. Furthermore, since all ties are broken in a consistent manner, all SimulationEvents bubble to the top of the priority queue in a consistent manner, at which time they dip into a stream of pseudorandom numbers provided by its ModelBlock's dedicated RNG.

As mentioned previously, the periodic bulk communication scheme is a key enabler of reproducibility. This periodicity ensures that events (OTCEs) from external threads are incorporated into a thread's priority queue at regular intervals, thus ensuring that they exhibit the consistent bubbling properties that are required for reproducibility.

Lastly, results cannot be recreated if the number of nodes or threads in the system changes between runs. Nor will results match when ModelBlocks are allocated amongst the GSAMs resources differently. And it goes without saying that no parameters should be changed between runs.

5. MEMORY USAGE

When the GSAM creates a simulated population, seemingly insignificant differences in an agent's memory-footprint can significantly impact overall memory usage. Saving one byte per person in a 6-billion-agent model saves 6 gigabytes of memory. Similarly, using an int where only a short is needed wastes 12 gigabytes of memory. Obviously, the GSAM code must be very careful to conserve memory when building its population.

The first impulse is to define a simple agent class like the SimpleAgent class as follows. Defining this class is the natural object-oriented (OO) thing to do. This class endows each agent with an age, a disease status, a family, and a reference to his schedule. The status variable is a reference to an Enum that defines the range of possible disease states. For example, a DiseaseStatus Enum could include the states Susceptible, Infected, and Recovered. The schedule variable is kept to enable swift removal of an agent's AgentEvent from a thread's Priority Queue when required. If this reference were not kept, removing an agent's AgentEvent from the Priority Queue would require an $O(n)$ search and an $O(\log(n))$ removal as opposed to just an $O(\log(n))$ removal [Goodrich and Tamassia 2008].

```
/* An example class -- Over simplified for brevity. */
public class SimpleAgent {
byte age;
DiseaseStatus status;
List_of_SimpleAgents familyMembers;
AgentEvent schedule;
}
```

The Simple Agent class is easy to understand. It encourages swift, error-free coding. It also wastes about 90% of the memory it uses! Using a "Simple Agent" class will force a model to use at least 49 gigabytes of memory to store a population of 1 billion agents (294GB for a population of 6 billion agents). Notice, this 49GB does not include space for any AgentEvents, any OffThreadContactEvents, or the entire contact matrix. The approach above is wasteful because the mere act of object creation wastes memory. All Java objects require more memory than their component variables alone require. This difference can be considered an objects memory overhead. The overhead includes at least one reference (without at least one active reference, the JVM will assume an object is garbage and reclaim that memory). The GSAM avoids the preponderance of this overhead by storing the agent data in large arrays. Example code is shown here.

```
/* Stores what a SimpleAgent[] would using less memory. */
public class AgentGroup {
byte[] ages; //length = numAgents
byte[] statuses; //length = numAgents
byte[] leftFamily; //length = numAgents
AgentEvent[] schedules; //length = numAgents
}
```

In this code, the ith agent is not a distinct SimpleAgent object. Rather, it is defined by the ith entry in each of the arrays in the AgentGroup object. This simple restructuring enables two additional sizeable reductions in memory costs beyond the elimination of object overhead.

The first reduction in memory cost concerns the storing of family members. In the SimpleAgent class, an 8-byte reference to a list of family members was kept (a 64-bit computing environment is assumed). The AgentGroup class can store that same information in the leftFamily field when all family members are stored in adjacent array entries. The leftFamily field stores the number of agents to the left of the ith agent that are family members of that agent. A list of each agent's family members can easily be computed from this array. As Figure 5 shows, the first member of each family has his "leftFamily" value set to 0.

A second memory savings (not implemented in the AgentGroup code, but included in Figure 5) can be had by replacing the AgentEvent array in AgentGroup with a hash table. Since hash tables normally have an array inside them, the benefit of this modification is not immediately clear. Recall, only infected agents have dedicated AgentEvents. Therefore, at any given time, most of the entries in the "schedules" array will be null. The GSAM saves memory by using a hash table to store only the nonnull references. This modification can reduce the cost of storing scheduling object references by roughly 66%. The exact memory savings will depend on the hash tables load factor and the proportion of agents who might be active at any one time. Note that using a standard off-the-shelf hash table is ill-advised because it will store 8-byte pointers and require Integer object wrappers – it is preferable to use a simple implementation to directly store 4-byte ints.

Another memory-saving technique is shown in the transition from the SimpleAgent class to the AgentGroup class. The "status" field in SimpleAgent was replaced with a "statuses" array in AgentGroup. However, notice that "status" is an object reference while "statuses" is an array of bytes. Here, we are storing a 1-byte number to indicate the disease status of an agent rather than an 8-byte reference to the appropriate Enum object (i.e., store *(byte) anEnum.ordinal()* instead of anEnum). This last improvement could have been incorporated into the original SimpleAgent class. However, this improvement violates the OO spirit of the SimpleAgent class.

Incorporating all of these memory saving improvements allows a population of billion agents to be stored in roughly 5.667GB as shown in Table II. This represents memory savings of 90%.

6. PERFORMANCE

6.1. Toward a Standardized Performance Metric

As briefly discussed previously, the universe of agent-based models is extremely diverse. Models study different topics, are written in different languages, and use different data structures. It would be helpful to have one common way to measure the performance of

these vastly different models. Having a common performance metric should facilitate discussions of modeling efficiency. Since agent-based modeling revolves around simulating behavior, we suggest measuring the number of individually implemented behaviors per second or BPS. This BPS benchmark will vary according t hardware capabilities, but it should still be an excellent starting point for comparisons between similar models.

BPS measurements will be lacking without a clear definition of “behavior”. W do not propose a general definition for behavior because relevant behaviors should vary wildly depending on the problem being modeled. For instance, ABMs of traffic flow might consider moving one car-length to be a single behavior while ABMs of disease transmission might consider person-to-person contacts to be the only relevant behavior. We believe that most applications of agent-based modeling will have natural definition for behavior – and by extension a natural BPS metric.

In this model of disease transmission we define behavior as:

- any person-to-person contact (family, coworker/classmate, and random contacts),
- any change in disease status (e.g., Susceptible to Noncontagious-Asymptomatic).

6.2. Benchmarks

All of results shown here are from simulations in which the following holds.

- The simulated day is broken into 100 discrete time periods.
- A simulated global population is used (LandScan 2007).
- There are 27,500 ModelBlocks (which are 20 km × 20 km squares):
  - **a.** ModelBlock populations are scaled down to reach the desired total population;
  - **b.** ModelBlocks are assigned to nodes in a round-robin fashion;
  - **c.** Each ModelBlock has summary data recorded each communication period.
- The simulated disease is influenza H1N1.
- The minimum communication time is 24 hours (or 100 time periods).
- The epidemic ends due to herd-immunity (when 36.6% have been infected).
- Agents perform, on average, 32.99 behaviors while active.

Our chief claim is that the GSAM is capable of simulating a global-scale epidemic in a reasonable amount of time. In Table III, we summarize the performance of a 32-node (one thread per node) GSAM run with a simulated population of 6.57 billion agents.

This particular run required 224GB of memory and 470.8 minutes (7.845 hours) to simulate a total of 2.40 billion infections in a population of 6.57 billion agents. Notice, this GSAM run communicates four times more frequently than required and that the communication accounts for approximately 27.6% of the total execution time.

Table IV is intended to show how the GSAM’s execution time varies with communication frequency. This data shows that communication frequency has a small effect on overall performance. If communication is put off as long as possible (24 hours, or 100 time periods,

for this particular pathogen), then a typical run takes 15.7 minutes (942 sec). However, if communication occurs 10 times more frequently, then runs take 20.1 minutes (1203 sec) - a 27.7% performance penalty. Earlier comments about the importance of bucketing still apply. But “frequent communication” in this context still reduces the total number of RMI transmissions by several orders of magnitude.

Scaling is an important property of any parallel program. The next two table illustrate how the GSAM scales across two dimensions. Table V shows how the GSAM scales up as the overall simulation repeatedly doubles in size. The next table shows how the execution time of a fixed simulation changes as the available computing power repeatedly doubles.

In general, the GSAM scales up quite well. However, we must be careful when interpreting the results in Table V and Table VI. Performance is affected by the amount of memory available. When there is more memory available, garbage collection (GC) occurs less often, resulting in better performance numbers. With this in mind, interpret the results below with a grain of salt, because all of these runs allocate 7GB of memory to each node regardless of whether that much memory is actually required.

From Table IV, we see that doubling the available computing resources will reduce the required execution time by 25–45%. This is not close to the 50% reduction that perfect scaling would achieve, but it is acceptable given that 32 well-equipped nodes can simulate an epidemic in a population of 6.57 billion agents.

6.3. Technical Details

One downside of Java is that maximizing the performance of any particular application typically requires a careful exploration of many different JVMs and the various options offered by each. For instance, all of the runs shown here used an option (actually two options used in tandem) that set the initial heap size equal to the maximum heap size. Had this option not been used, the measured execution time would have included several tremendously expensive, on-the-fly heap expansions. These heap expansions degrade performance, and confuse performance measurements.

One particular way in which JVMs vary is in their implementation of garbage collection. The JVM used herein uses a “stop-the-world” approach to GC, meaning that when the JVM starts garbage collection, it temporarily stops all other threads. This approach to GC seriously inhibits the performance of nodes with many threads. Consequently, even though the GSAM allows nodes to run many threads, performance is maximized when each node executes exactly one thread (at least it is when using the Java Hotspot Server VM (1.6.0 07) that comes bundled with the 1.6.0 07 JDK).

Everything was compiled and run using the 64-bit edition of Java 1.6.0 07. It is important to note that performance can vary according to the JVM in use. Consequently, even though the numbers shown are quite good given the scale of the problem, it may be possible to improve them simply by swapping out the JVM.

All the GSAM performance numbers shown were produced on as many as four HP DL585 servers. Each of these is a quad processor server equipped with 64GB of memory and a full allotment of Opteron 8216 (2.4GHz) dual-core CPUs. In total, this system has 32 CPU cores and 256GB of memory. Each of the servers runs Windows Server 2003 Enterprise x64 Edition.

7. CONCLUSION

To our knowledge, the GSAM is the first bona fide agent-based model that can efficiently execute global-scale infectious disease simulations, those involving several billion distinct agents. The GSAM appears also to be the first agent-based model that can execute a simulation of several hundred million agents in a matter of minutes. Although this exposition has been couched in terms of infectious diseases, it is clear that the GSAM platform proper can be applied to a wide range of social, economic, and public health problems of global scope. These capabilities derive from the particular distributed parallel approach that was adopted in the design phase. The resulting high ceiling allows researchers to implement whatever population best suits their research needs.

Not only does the GSAM enable more efficient modeling, but it should also enable more collaborative modeling. Previously, large ABMs were likely to be effectively tied to a specific set of hardware. It is too easy to “tune” a massive message-passing interface model so much as to undermine deployment on different hardware. Since the GSAM is written in pure Java, it is perfectly portable. Also, since it makes very efficient use of its memory, all but high population (50 Million or more) models could be deployed on high-end workstations. Combined, these two features could allow research teams to work faster because a model-in-progress can be executed in multiple places, not just on one specific hardware platform.

In sum, the GSAM opens the door to flexible, efficient, portable high-resolution agent-based modeling on a truly global scale.

Acknowledgments

For detailed comments on the prepublication draft, the authors thank Philip Cooley and Diglio Simoni of Research Triangle Institute. We also acknowledge the invaluable comments and suggestions of anonymous reviewers, and the journal’s editors.

This project was supported by The DHS National Center for the Study of Preparedness and Catastrophic Event Response (PACER) at the Johns Hopkins University through Award Number N00014-06-1-0991 from the Office of Naval Research, The NIH Models of Infectious Disease Agent Study (MIDAS), Award Number U01GM070708 from the National Institute of General Medical Sciences, The University of Pittsburgh Public Health Adaptive Systems Project (PHASYS) through Cooperative Agreement Number 1P01TP000304-01 from the Centers for Disease Control and Prevention on the National Science Foundation Grant Number SES-0729262 Collaborative Research, Modeling Interaction between Individuals Behavior, Social Networks and Public Policy to Support Public Health Epidemiology. J. M. Epstein acknowledges support from the 2008 Director’s Pioneer Award Number DP1OD003874 from the Office of the Director, National Institutes of Health. The content of this article is solely the responsibility of the authors and does not necessarily represent the official views of any funding organization.

Appendix A

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

Anderson, R.; May, R. Infections Disease of Humans: Dynamics and Control. Oxford Sciences Publications; 1993.

Bergstrand JH. The gravity equation in international trade: Some microeconomic foundations and empirical evidence. Rev. Econ. Stat. 1985; 67:474–481.

Burke DS, Epstein JM, Cummings DAT, Parker JI, Cline KC, Singa RM, Chakravarty S. Individual-based computational modeling of smallpox epidemic control strategies. Acad. Emerg. Med. 2006; 13(11):1142–1149. [PubMed: 17085740]

Carothers C, Fujimoto R. Efficient execution of time warp programs on heterogeneous, NOW platforms. IEEE Trans. Paral. Distrib. Syst. 2000; 11(3):299–317.

Christopher, T.; Thiruvathukal, G. High Performance Java Platform Computing. Prentice-Hall; 2001.

Cooley P, Ganapathi L, Ghneim G, Holmberg S, Wheaton W. Using influenza-like illness data to reconstruct an influenza outbreak. Math. Comput. Model. 2008; 48(5–6):929–939. [PubMed: 19122846]

Deelman, E.; Szymanski, K. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems; Proceedings of the 12th Workshop on Parallel and Distributed Simulation; 1998. p. 46-53.

Epstein, JM. In Ethical and Legal Considerations in Mitigating Pandemic Disease, Workshop Summary. Institute of Medicine. The National Academies Press; 2007. Remarks on the Role of Modeling in Infectious Disease Mitigation and Containment; p. 105-112.

Epstein, J.; Axtell, R. Growing Artificial Societies: Social Science from the Bottom Up. Brookings Institute Press; 1996.

Epstein, JM.; Cummings, DAT.; Chakravarthy, S.; Singa, RM.; Burker, DS. Toward a Containment Strategy for Smallpox Bioterror: An Individual Based Computational Approach. Brookings Monographs; 2004.

Epstein J, Parker J, Cummings D, Hammond R. Coupled contagion dynamics of fear and disease: Mathematical and computational explorations. PLoS ONE. 2008; 3:12.

Eubank S, Guclu H, Anil Kumar VS, Marathe MV, Srinavasan A, Toroczkai Z, Wang N. Modeling disease outbreaks in realistic urban social networks. Nature. 2004; 429:180–184. [PubMed: 15141212]. [PubMed: 15141212]

Ferguson NM, Cummings DAT, Cauchemez S, Fraser C, Riley S, Meeyai A, Iamsirithaworn S, Burke DS. Strategies for containing an emerging influenza pandemic in Southeast Asia. Nature. 2005; 437:209–214. [PubMed: 16079797]. [PubMed: 16079797]

Ferguson NM, Cummings DAT, Fraser C, Cajka JC, Cooley PC, Burke DS. Strategies for mitigating an influenza pandemic. Nature. 2006; 442:448–452. [PubMed: 16642006]. [PubMed: 16642006]

Fujimoto RM. Parallel discrete event simulation. Communications of the ACM. 1990; 33(10):30–53.

Goodrich, M.; Tamassia, R. Data Structures and Algorithms in Java. 5th Ed. Wiley; New York, NY: 2008.

Gosling, J.; Joy, B.; Steele, G.; Bracha, G. The Java Language Specification. 3rd Ed. Addison Wesley; 2005.

Glass RJ, Glass LM, Beyeler WE, Min HJ. Targeted social distancing design for pandemic influenza. Emerg. Infect. Dis. 2006; 12(11):1671–1681. [PubMed: 17283616]. [PubMed: 17283616]

Isard W. Location theory and trade theory: Short-run analysis. Quart. J. Econ. 1954; 68:305–322.

Java Remote Method Invocation Documentation (Java SE 6). http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html

Java Serialization Documentation Java SE 6. http://java.sun.com/javase/6/docs/technotes/guides/serialization/

Kermack W, McKendrick A. A contribution to the Mathematical Theory of Epidemics. Proc. Roy. Soc. London A Mat. 1927; 115:700–721.

Keeling, MJ.; Rohani, P. Modeling Infectious Diseases in Humans and Animals. Princeton University Press; 2007.

Kremer M. Integrating behavioral choice into epidemiological models of the AIDS epidemic. Quart. J. Economics. 1996; 111:549–573.

Lee S, Wilson J, Crawford M. Modeling and simulation of a nonhomogeneous Poisson process having cyclic behavior. Comm. Stat. Simul. 1991; 20:777–809.

Longini IM, Halloran ME, Nizam A, Yang Y. Containing pandemic influenza with antiviral agents. Amer. J. Epid. 2004; 159:623–633. [PubMed: 15033640].

Longini I, Nizam A, Xu S, Unqchusak K, Hanshaoworakul W, Cummings DA, Halloran ME. Containing pandemic influenza at the source. Science. 2005; 309:1083–1087. [PubMed: 16079251]. [PubMed: 16079251]

Mathew, A.; Roulo, M. Accelerate your RMI programming. 2003. http://www.javaworld.com/jw-09-2001/jw-0907-rmi.html?page=3

Peschlow, P.; Honecker, T.; Martini, P. A flexible dynamic partitioning algorithm for optimistic distributed simulation; Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation; 2007.

Perumalla, K. Parallel and distributed simulation: Traditional techniques and recent advances; Proceedings of the 38th Conference on Winter Simulation; 2006.

Robert, CP.; Casella, G. Monte Carlo Statistical Methods. 2nd Ed. Berlin: Springer-Verlag; 2004.

Voorhees, A. A general theory of traffic movement; Proceedings of the Institute of Traffic Engineers; 1956.

Simon, AH. Models of Bounded Rationality. Cambridge, MA: MIT Press; 1982.

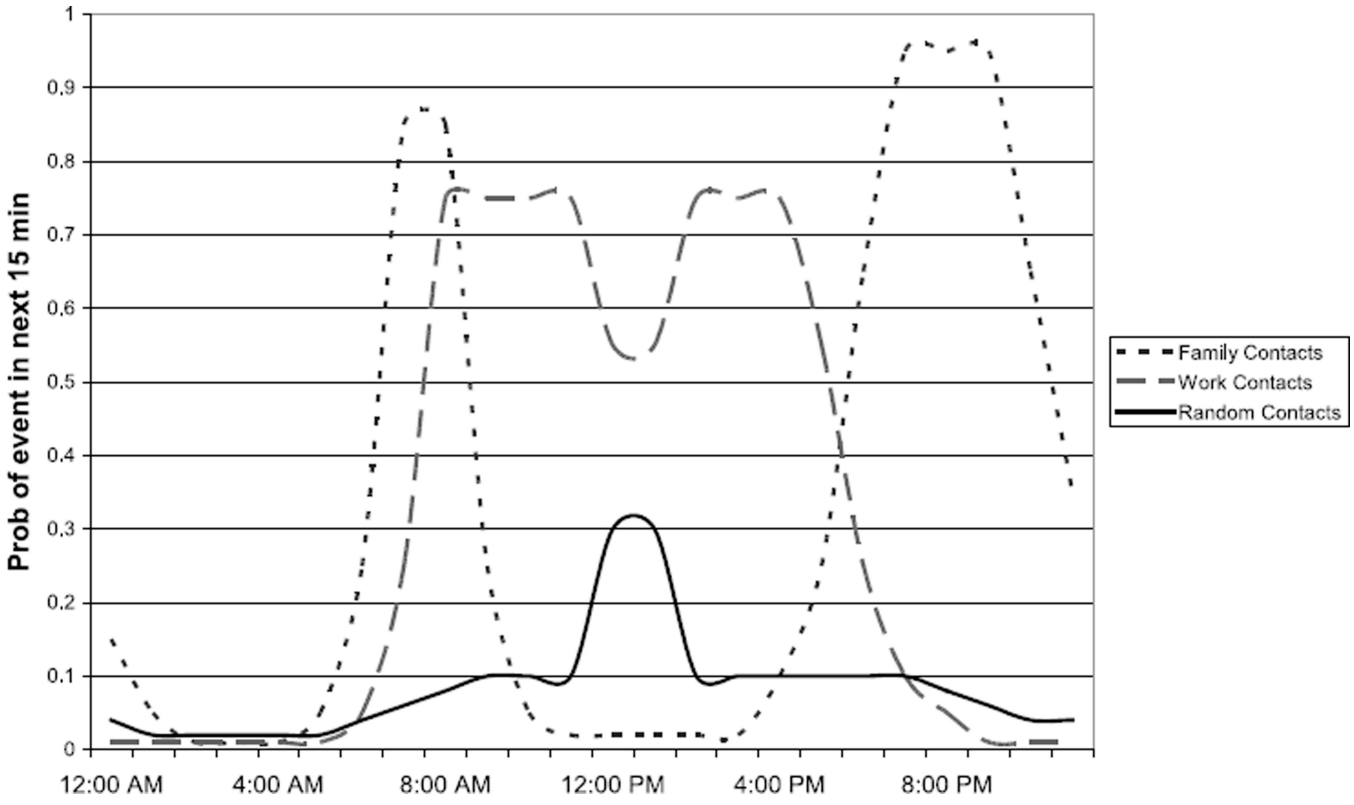

Fig. 1.

Each line in this graph depicts a plausible behavior stream. Each behavior stream determines the probability that a specific event type is included in an agent's itinerary at some future time. Notice, this agent is less likely to make coworker contacts at lunch time.

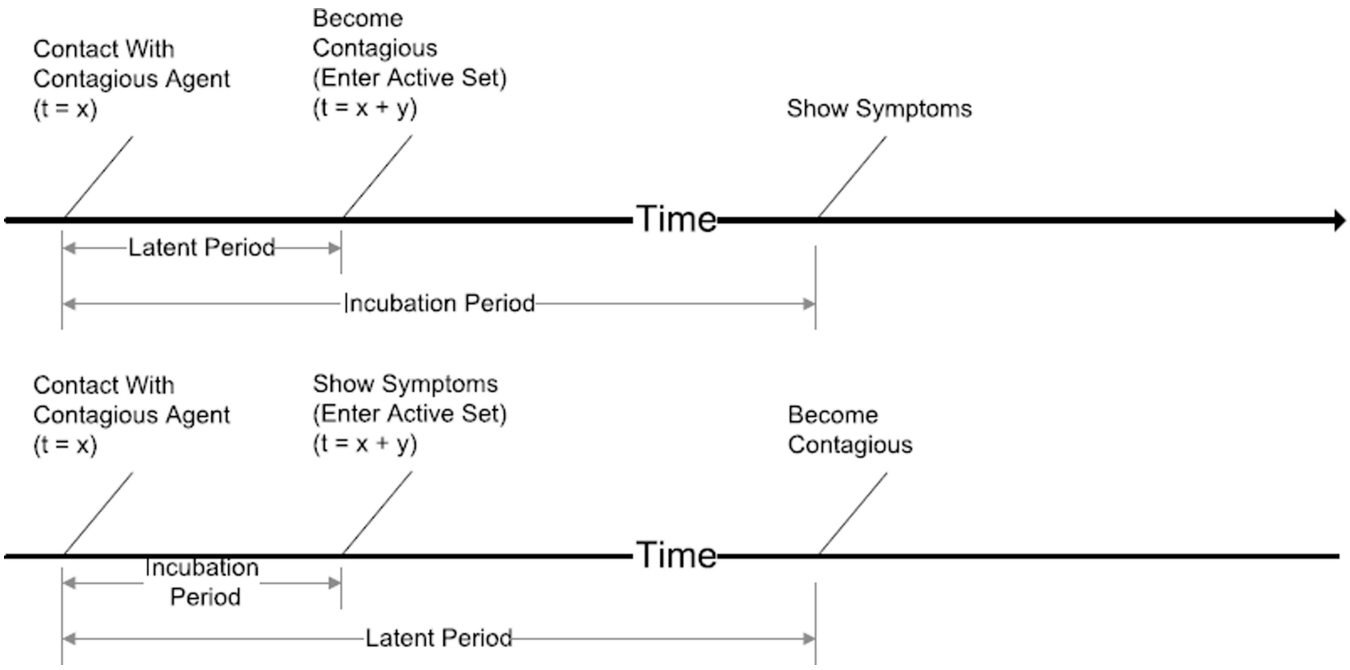

Fig. 2.
These two timelines show how the latent and incubation periods are measured.

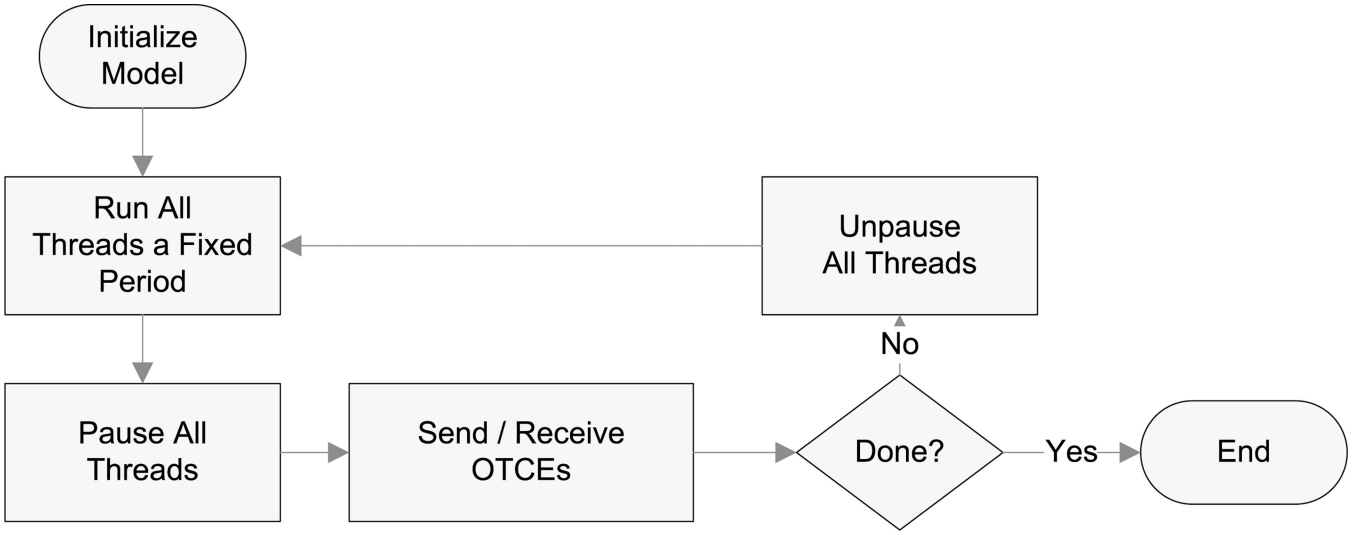

Fig. 3.
A basic flowchart of the progress of a GSAM run.

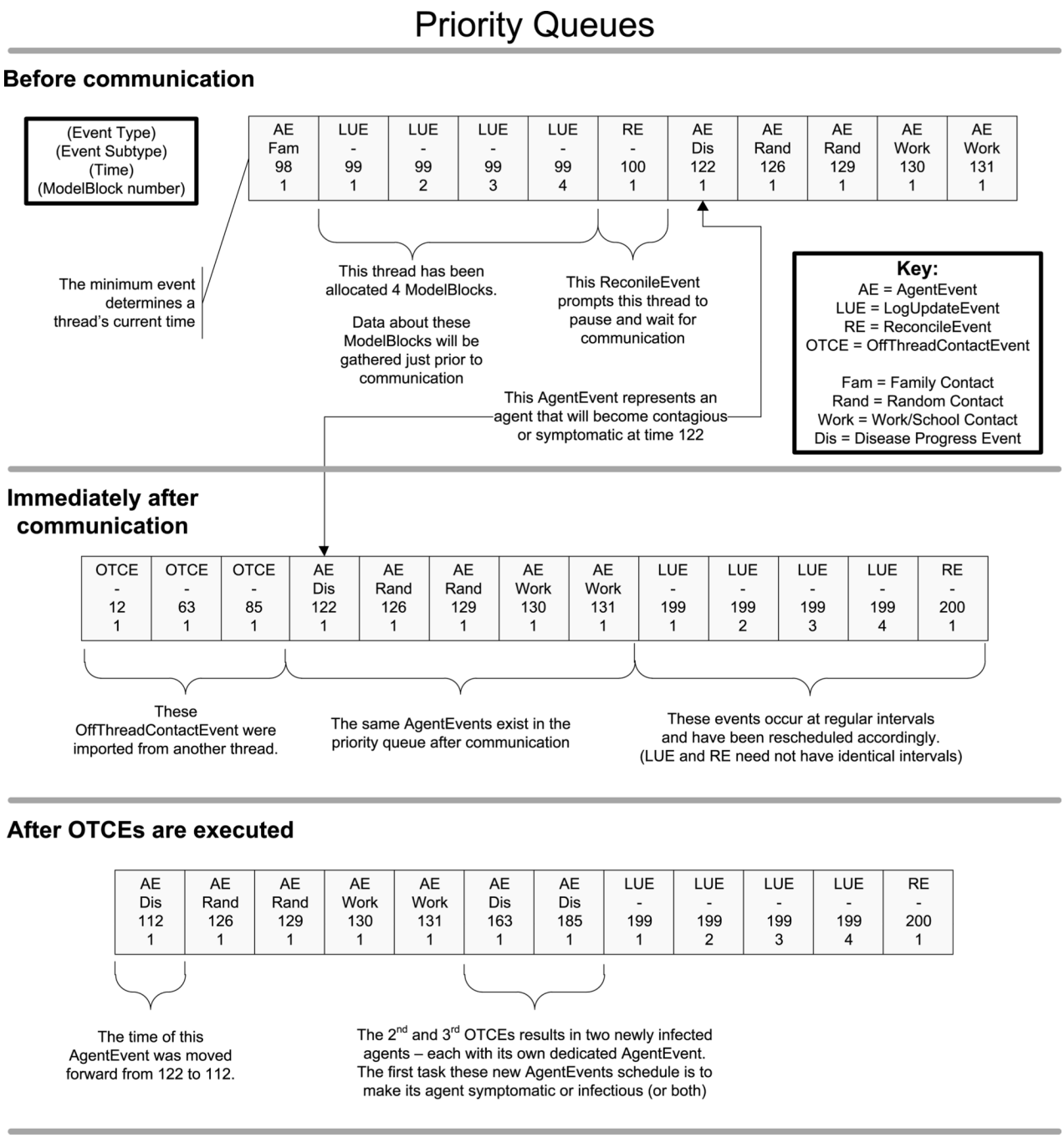

Fig. 4.
This diagram illustrates many aspects of GSAM implementation.

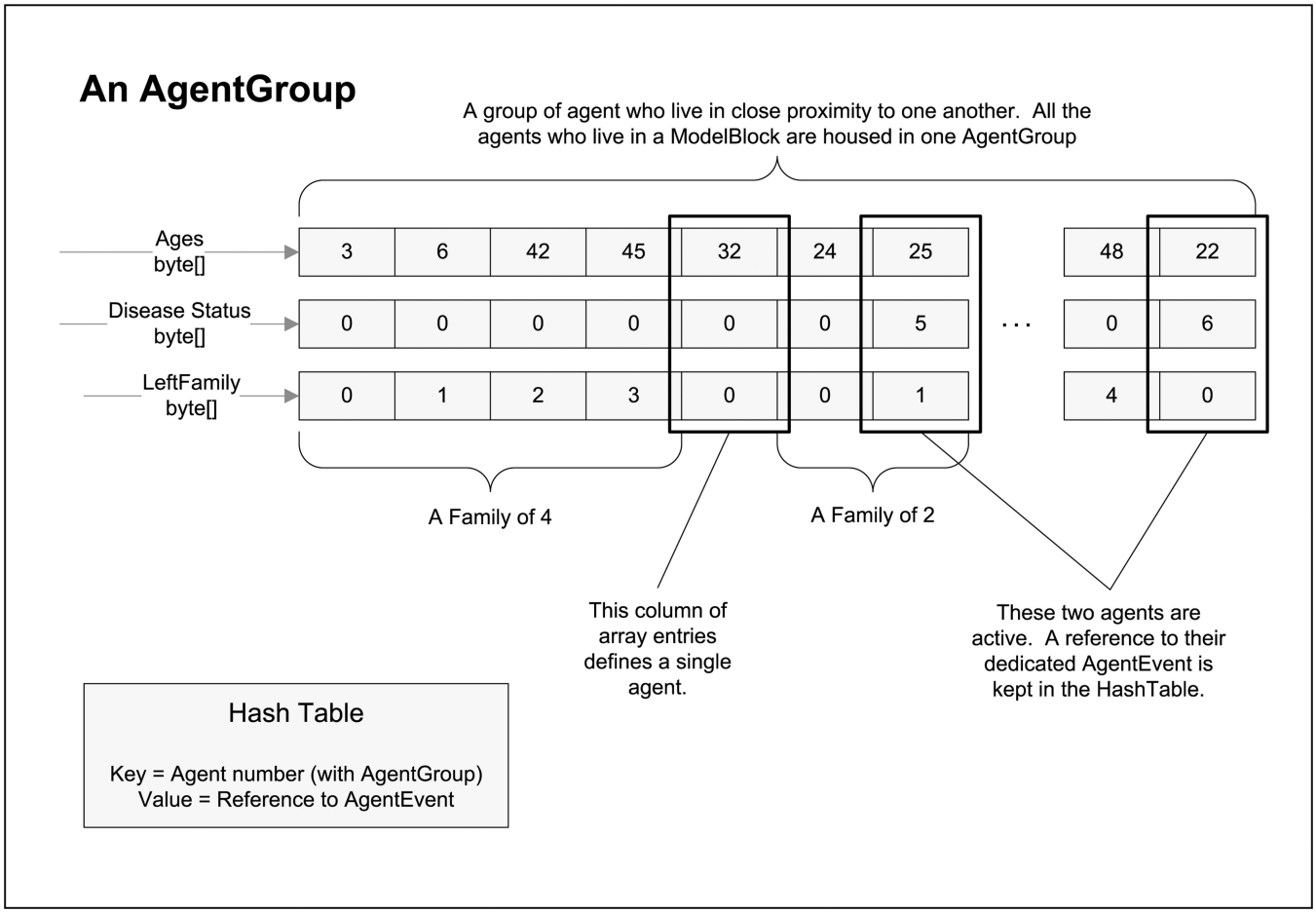

Fig. 5.

A graphical representation of an AgentGroup.

NOTE: A realistic AgentGroup class should contain more arrays. This depiction does not even include gender.

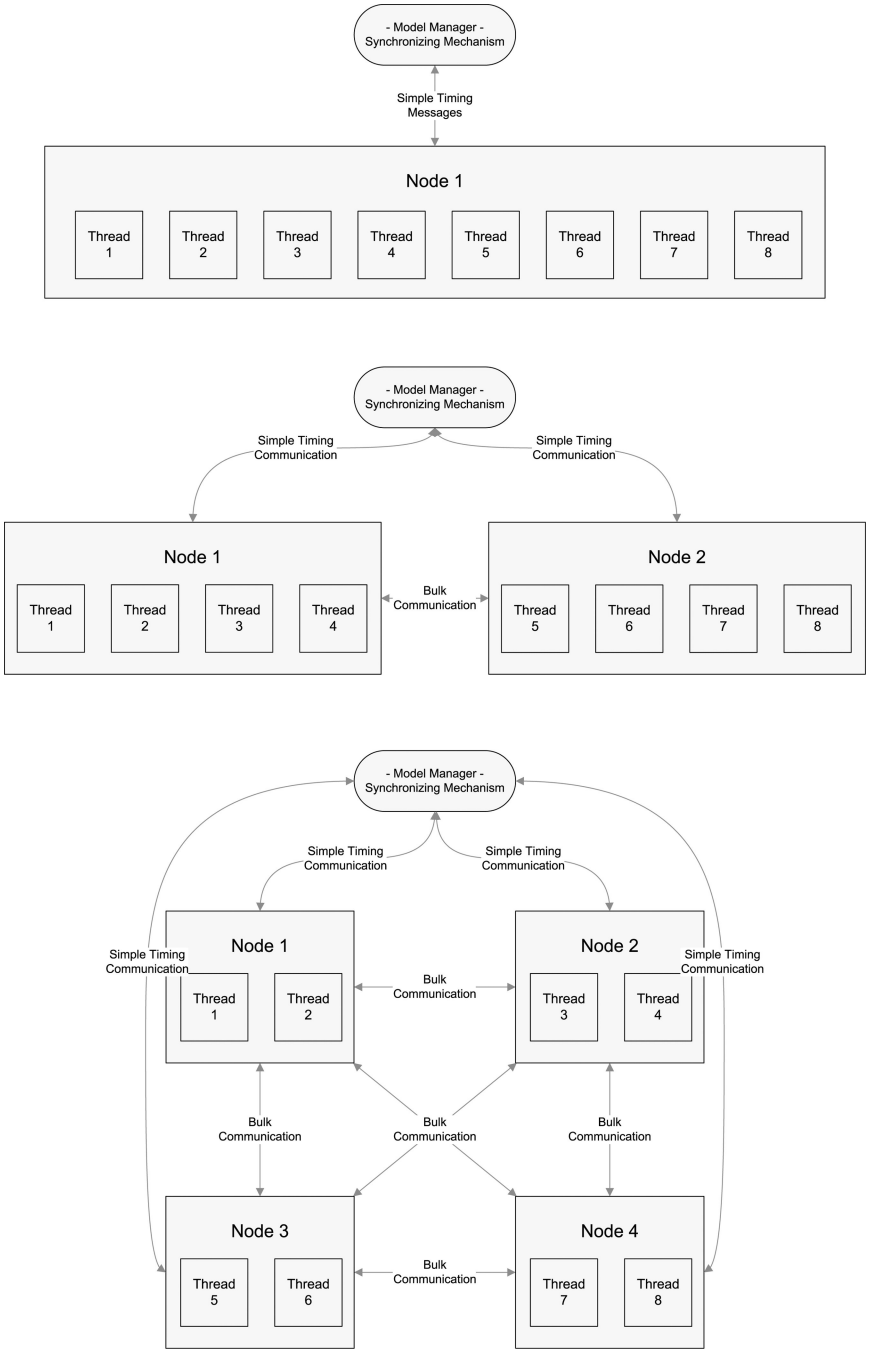

Fig. 6.
Three GSAM set ups intended to use the same number of CPUs.

Table I

Time and Operations Required to Send 10 Million Doubles between Two JVMs Using RMI Methods with Different Signatures

Actual RMI Method	Number of RMI Calls Required	Number of ser/deser Pairs Req.	Average Time (sec)
send(Double d)	10,000,000	10,000,000	858.330
send(double d)	10,000,000	0	637.853
send(Double[] ds)	1	10,000,000	23.433
send(double[] ds)	1	0	1.141

Table II

Memory Costs per Agent

	SimpleAgent Class	**AgentGroup Class**
Object Overhead	16 bytes	N/A
A Required Reference	8 bytes	N/A
Age	1 byte	1 byte
Disease Status	8 byte	1 byte
Family Members	8 bytes	1 byte
Scheduling Object	8 bytes	2.667 bytes
Total	49 bytes	5.667 bytes

Table III

A Global Scale Simulation

Population	Nodes	Total Execution Time (min)	Total Communication Time (min)	BPS	Memory Per Node	Communication Freq
6.57 Billion	32	470.8	129.9	2,809,181	7GB	25

Table IV

Summary of Communication Frequency vs. Execution Time

Population	Nodes	Total Execution Time (sec)	Total Communication Time (sec)	Behaviors Per Second (BPS)	Communication Freq
98,577,284	8	1203	197	989,310	10
98,577,284	8	1062	179	1,120,979	20
98,577,284	8	994	172	1,197,681	40
98,577,284	8	975	167	1,220,816	60
98,577,284	8	966	165	1,231,999	80
98,577,284	8	942	158	1,263,496	100

Table V

Weak Scaling (Constant Population per Node)

Population	Nodes	Total Execution Time (sec)	Total Communication Time (sec)	Behaviors Per Second (BPS)	Comm Freq
65,718,190	1	2991	0	265,252	100
131,436,379	2	3668	460	432,615	100
262,872,758	4	4321	833	734,521	100
525,745,516	8	4701	964	1,350,501	100
1,051,491,033	16	6003	1692	2,114,952	100
2,102,982,066	32	7369	2119	3,445,989	100

Table VI

Strong Scaling (Constant Population)

Population	Nodes	Total Execution Time (sec)	Total Communication Time (sec)	Behaviors Per Second (BPS)	Comm Freq
98,577,284	1	4283	0	277,915	100
98,577,284	2	2844	367	418,485	100
98,577,284	4	1518	251	783,939	100
98,577,284	8	962	160	1,237,915	100
98,577,284	16	606	116	1,965,415	100
98,577,284	32	425	75	2,801,920	100