Proceedings of the 2007 Winter Simulation Conference
S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds.

A FLEXIBLE, LARGE-SCALE, DISTRIBUTED AGENT BASED EPIDEMIC MODEL

Jon Parker

Center on Social and Economic Dynamics
The Brookings Institution
Washington DC 20036, U.S.A.

ABSTRACT

We describe a distributed agent based epidemic model that is capable of easily simulating several hundred million agents. The model is adaptable to shared-memory and distributed-memory architectures. Several problems are addressed to enable the distributed simulation: allocation of agents to available compute nodes, periodic synchronization of compute nodes, and efficient communication between compute nodes. We assert that our modeling scheme is easily adaptable to different hardware environments and does not require large investments in performance tuning or special case coding.

1 INTRODUCTION

We set out to create an agent based model (ABM) of disease transmission. We designed our model to suit multiple research projects, collaborators, and computing facilities. Consequently, our model had to satisfy many different requirements. The model had to be capable of supporting different diseases. It also had to support two different computing environments. One environment was a 64 bit Linux-based distributed-memory cluster with low latency interconnects. The other environment was a pair of Windows-based quad processor servers each with 32 GB of shared-memory. Our last major requirement was to be able to support populations from several hundred million to 6 billion agents.

In this paper we will discuss the overall design of the model. We will also explicitly cover three topics. We will detail the synchronization of multiple computers, and the allocation of agents to those computers. We also dedicate a section to the implementation of the communication protocol. This last section is the only language specific section in the paper.

2 OVERALL DESIGN

First of all, the entire model was written in JAVA, however only a small fraction of this paper is JAVA specific. Implementing the model in JAVA allows us to quickly and easily step away from the problems supporting different operating systems might pose. JAVA will also allow development to proceed swiftly due to the language's built in features such as extensive libraries, automated serialization, and Remote Method Invocation (RMI).

The most immediate problem to address when designing a large scale ABM is that the entire population of agents will not fit onto one node of a distributed-memory cluster. Therefore, a scheme to distribute the agents across available computing resources must be devised. Our scheme distributes the agents across the computing resources in two phases. First, the agents are distributed amongst the computers that will be running the model. Then agents are distributed between the threads on each computer. Each computer will host one processing thread for each CPU it has (2 or more threads per multi-core CPU).

Communication between threads and nodes only occurs at prearranged times. When a thread reaches a communication time it is paused. The thread will then send messages to other threads via the "node" program described below. After the outgoing messages are sent incoming messages are received. All messages that need to be sent are stored in a queue until a prescheduled communication time is reached, then all messages are sent as a single unit. Queuing the communication in this way removes the communication bottleneck derived from RMI's high latency.

The final two pieces of the model are small programs that oversee the model as a whole. An instance of a "node" program is run on each computer that runs a fraction of the model. The node program manages the threads of a single computer. The second program is the "manager" program. The manager program ensures that all communication has finished before it allows each node to resume independent processing. The manager also functions as the hub of all communication. There is only one instance of the manager program.

3 SYNCHRONIZATION DETAILS

In this section, we discuss precisely how threads navigate a loop from processing, to waiting, to communicating, and back to processing again. To start we will describe the threads themselves. At the heart of each processing thread is a single special purpose Priority Queue (PQ). This PQ must support efficient removal of a random entry. This is often an O(n) operation because most PQs are implemented with a heap or an array. To create our special purpose PQ we added a method named "removeFirst(key)" to the java.util.TreeMap class which is implemented using a Red-Black tree. Combining the remove(key) and firstKey() into one method significantly improves performance.

Priority Queues contain events that are tagged with a time. Normal processing proceeds when a thread polls the event with minimum time off of the PQ, executes it, and repeats. Eventually a processing thread will poll an event that denotes the scheduled communication time has been reached. The thread is then paused until all communication is complete.

Once every thread on a node is paused that node is ready to communicate with the manager. But what exactly is communicated? Well, when the processing thread was polling events occasionally it would encounter an event that required communication with an agent that resides on another thread or node. Since the required communication won't occur until later that event could not have been fully executed immediately. A record of this unexecuted event was placed in a queue for later execution. These records are coined OffNodeContactEvents (ONCE). These ONCEs are the bulk of the inter-node and inter-thread communication.

At first glance the process of saving some events to be executed later appears severely flawed. Here is an example of the obviously problem. Let ONCE *alpha* represent a contact between two agents that should have occurred at time 172. Yet, *alpha* gets transmitted and executed at time 200. How has this not corrupted the timeline of the model?

The answer can be found in the incubation period of the disease we are simulating. If an agent is exposed and infected with a disease at time period x but does not start changing behavior or infecting others for y time periods then the record of that agent's infection doesn't need to be sent until a time period just before $(x + y)$. The maximum time between communication periods is dictated by the incubation period of the disease being simulated. When communication does occur there are only two pieces of bookkeeping that must occur. First, ensure that any newly infected agent was infected at the time the ONCE was created and not when the ONCE was sent or executed. Second, occasionally one must alter an agent's "infection time". For Instance, push back an agent's "infection time" if that agent was infected at time 188 but should have been infected by a ONCE at time 172. Making this correction requires searching the Priority Queue for the specific event that marks the beginning of an agent's infectious period. If the PQ was implemented with a heap or an array this operation's inefficiency would noticeable impact performance.

When all nodes are ready to communicate the manager program retrieves the queue of OffNodeContactEvents from each node. The manager will then distribute these ONCEs to the appropriate node. Next, the manager will confirm that all nodes have finished incorporating the new OffNodeContactEvents into the appropriate thread's Priority Queue. At this point the manager prompts each node to notify its threads to begin processing again.

4 DETAILS OF COMMUNICATION IMPLEMENTATION

Of course the communication is implemented using JAVA RMI. We know that RMI employs other built-in JAVA features to work its magic so we must be careful. In particular, we are trying to avoid nesting too many "general purpose" built-in capabilities so that performance doesn't degrade.

JAVA RMI converts objects it sends over the network to binary data. RMI then sends the binary data over the network. Upon receiving the binary data it reconstructs those objects from the binary data. Converting an object to binary data is called serialization. Reconstructing the object is called deserialization. These two processes are automated by the JAVA language. However, this automation is very expensive time-wise.

There are two ways to reduce the cost of serialization calls. One, we can bucket many OffNodeContactEvents into one object. This saves an enormous amount of time because now we only call the built-in serialization scheme once per bucket. Two, rather than rely on JAVA to generate serialization and deserialization methods we can explicitly code them. To do this implement the Externalizable interface in the java.io package. Be aware that implementing Externalizable will not yield a large performance improvement if a bucketing scheme is in place.

Lastly when running the model with many node programs it is very important to communicate in parallel. Retrieving ONCEs from the nodes is a quick process because those objects have already been created and bucketing is swift just like communication. When retrieving, it barely matters if retrieval occurs in serial or in parallel. However, sending ONCEs back to the nodes appears much slower. But, very little time is spent on the actual communication. The majority of the time is spent incorporating the new OffNodeContactEvents into Priority Queues. Since the node programs and their Priority Queues are all independent that work should be done in parallel to save time.

5 DISTRIBUTION OF AGENTS BETWEEN COMPUTING RESOURCES

We want to distribute the agents across our computing resources such that the model runs as fast as possible in real time. There are two important facts to remember when designing a distribution scheme: a node's runtime will be closely associated with the number of infected agents residing on that node; contacts that require communication are dramatically more expensive then contacts that require only local information. These two competing forces suggest dramatically different distributions of agents. On one hand, assigning each node a 1/n share of each geographic region would suggest that all nodes equally share the workload. Thus promoting load-balancing and reducing the amount of time nodes sit idle waiting for other nodes to finish. On the other hand, we could assign an entire geographic region to each node. This would minimize expensive contacts that require communication. The first way increases the total amount of work to be done by maximizing communication. The second way maximizes the likelihood that some of our computing resources will be sitting idle due to a geographic specific disease outbreak.

After running many different empirical experiments we found that our model runs overwhelmingly faster when inter-node communication is minimized. Nodes must be significantly (perhaps unrealistically) unbalanced for a large fraction of the model run before it pays to split geographic regions between nodes and bloat the amount of inter-node communication.

Using the results of our empirical experiments as a guide, we can now start to formulate a strategy to optimize the distribution of our agents. The computing environment will dictate how many ways we must divide our population. For instance, if we intend to run our model on 2 quad processor servers then we need to divide our population into 8 groups. To be more precise, we need to first divide the population into 2 groups, one group for each node. Then we further divide each of those 2 groups into 4 subgroups.

Before we can discuss the details of our distribution optimization approach we should discuss the basic building block of our model. The basic building block represent a group of agents that all reside close to one another in geographic space. The building block could represent all the agents who live in a city block, a zip code, a census tract, or perhaps a square kilometer. We only require that we are able to measure the distance between each pair of building blocks. For the purposes of this paper let us refer to these building blocks as pixels. Now when we visualize this problem, we can think about it as coloring a map made up of n pixels.

To begin, we need to calculate some data. We assemble a list of each pixel's population. Then, we calculate an n by n matrix A where matrix entry A_{ij} is an estimation of the total amount of interaction between agents that live in pixel i and agents that live in pixel j. Armed with this data we can begin distributing our agents.

First randomly assign each pixel one of g possible colors, where g is the number of groups the population will be split into. After each pixel has been assigned a color we then create a small 2-dimensional array. This array has dimensions "number of pixels" by "number of groups". The entry array[i][j] is a measure of how much pixel i interacts with all pixels of group (color) j. This array can be directly calculated from matrix A once each pixel has been assigned a color. Maintaining this helper array throughout optimization will drastically improve performance. This is akin to storing the sum of a set of numbers to avoid recalculating the sum from scratch after one number is added or removed from the set.

Now we enter the core optimization loop. We pick a random group. That group absorbs pixels from other groups until it reaches a certain population. Then this group starts to shrink, giving pixels to other groups. When the shrinking group's population is about "total population / g" it stops donating pixels to other groups. Then pick another random group and repeat.

Each time the loop runs the selected group grows a little less. Eventually no pixels will be exchanged and the loop will exit leaving behind groups that do little inter-group interaction. This approach was loosely inspired by simulated annealing.

When a group is growing it must select which pixel it wants to steal. It will select the out-of-group pixel that it interacts with most. It does not matter if per capita interaction or total interaction is selected as the metric. Both ranking schemes will provide good results.

However, a shrinking group must be more careful when it decides which pixel to expel. A shrinking group should expel the pixel that does the smallest share of its interaction within the group. Expelling the pixel that does the smallest absolute amount of intra-group interaction almost always expels a small population pixel regardless of whether that pixel should rightfully be a member of the group.

A shrinking group must also decide which group receives expelled pixels. An expelled pixel's new owner should be the group that it interacts with most (that isn't the expelling group). Other methods of assigning new owners create noticeable problems. For instance, assigning pixels to a random group slows optimization, and leaving pixels as a member of no group leaves many pixels neglected because they never get absorbed by a growing group.

An obvious approach would be to run the above algorithm once to split n pixels into 8 groups. But it is a much better idea to run that algorithm multiple times to split those n pixels into 8 groups. Each time we run the algorithm we should split the starting population into as few

groups as possible (in this case 2). Nesting the optimization calls is better for two reasons.

One, it provides a better final distribution. The above algorithm works with many groups, but it does not work well with many groups. When there are too many groups (6+) there are opportunities for the core loop to start exhibiting cycle-like behavior. This cycle-like behavior might leave 6 of 8 colors compact and well optimized but leave the other 2 colors very disjointed.

Nesting optimization calls also reduces the total time required to calculate a good distribution. The two major determinants of our algorithm's runtime are "Matrix size" and "number of groups". Each time the algorithm runs it will be optimizing a quadratically shrinking matrix. Since our algorithm converges quadratically faster as it splits the population fewer ways it is best to optimize as quickly as possible when operating on the full size matrix. Any further population splitting will proceed quite quickly.

We do not claim that our distribution algorithm will minimize inter-group contacts. We make the weaker claim that our algorithm quasi-minimizes inter-group interaction. We estimate that exactly minimizing inter-group interaction could only further reduce inter-group interaction by at most 10 percent.

Finally, we present the results of optimizing the distribution of agents amongst two nodes where each node hosts eight threads. Figure 1 shows the distribution between the two nodes. Figure 2 shows how each node allocates its share of the model amongst its 8 threads. The similarly colored regions on opposite sides of the red dividing line are not related.

As you can see in Figure 1 the obtained distribution is almost certainly not the best possible result. Our algorithm has trouble swapping a region that is made up of multiple pixels. In this case the red Miami region did not swap to the green node. This is likely to be a small inefficiency. The same forces that kept the Miami region from being transferred to the green node pixel by pixel also ensured that there is a small penalty for this failure. The green node would have absorbed a pixel of the red Miami region if it interacted strongly with it. Had a single pixel of the red Miami region become green it would have encouraged a cascade that eventually would have absorbed every pixel of the red Miami region. Since this cascade did not occur we can conclude that the red Miami region does a very large share of its interaction within the local red region (hence not the surround green). If this is indeed true it is only important that the Miami region is not split. It is not important if the region is green or red, because it "only" interacts within itself.

Figure 1: The quasi-optimal distribution of agents between two nodes.

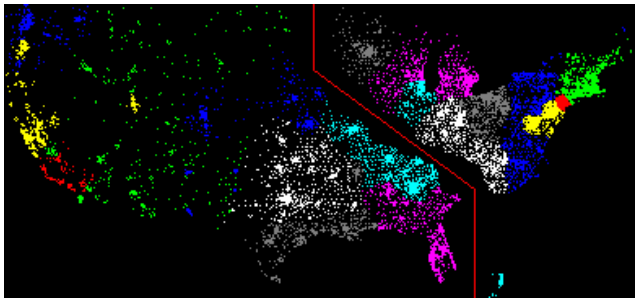

Figure 2: This figure shows the same distribution as in Figure 1 except the colors represent the distribution between the thread of each node.

Table 1: This table shows proof of performance numbers. Each simulation had a population of 300 million agents.

	Total Time (min)	**Percent time communicating**	**Communication Frequency**
Run A	131.68	14%	every 12 hours
Run B	119.32	15%	every 48 hours
Unoptimized	161.91	24%	every 12 hours

6 BENCHMARKS

In this section we show the results from three different simulations. The simulations were of a stylized disease spreading across the continental United States. The stylized disease had a 96 hour incubation period and a 48 hour contagious period. The model population reaches herd immunity when roughly 65% of the community has recovered from the disease. At this point the epidemic dies out due to a lack of susceptible agents. Each simulation lasts about 300 days and has a population of 300 million agents living in 4000 pixels using about 38 GB of memory. The numbers presented here are put forward as a proof of concept and performance.

One of the three runs did not have the allocation of agents to nodes and threads optimized. This run is included here to support our claim that reducing inter-node and inter-thread communication at the expense of load balancing is beneficial.

The other two simulations are better representations of achievable performance. Two observations are of particular importance. First of all, a detailed simulation of a 300 million agent population takes roughly 2 hours to complete and only uses 38 GB of memory. This leaves plenty of room for expansion along practically any dimension desired. Second, notice that the time spent communicating between computing resources is not a dominant factor in model run-time.

7 CONCLUSION

We believe JAVA is an excellent language with which to develop large agent based models. JAVA provides many built in features like RMI, serialization, and thread management tools that allow a distributed model to be easily developed. While there are important pitfalls to watch out for when using JAVA these obstacles can be easily sidestepped. Any modeler is then left with a wonderful toolbox to create a flexible portable, efficient, and clean epidemic model.

AUTHOR BIOGRAPHIES

JON PARKER is a senior research software engineer at The Brookings Institution. He hold a B.S. in Math Sciences from John Hopkins University. He has developed the agent model partly described here to support grants from the National Institutes of Health as well as the Department of Homeland Security.