Towards Simulating Billions of Agents in Thousands of Seconds

Tracking Number: 578

ABSTRACT
Building multi-agent systems that can scale up to very large number of agents is a challenging research problem. In this paper, we present Distributed Multi Agent System Framework (DMASF), a system which can simulate billions of agents in thousands of seconds. DMASF utilizes distributed computation to gain performance as well as a database to manage the agent and environment state. We discuss the design and implementation of DMASF in detail and present experimental results. DMASF is a generic and versatile tool that can be used for building massive multi agent system applications.

1. INTRODUCTION
Many Multi Agent Simulation Systems do not scale to a large number of agents. With dropping hardware costs, computer networks are present almost everywhere. Many of the Multi Agent Systems do not utilize the advantages of distributing the simulation work across multiple computers in a networked environment.

The ability to pause and resume complex simulations is something that is missing in most Multi Agent System simulators. This applies more to simulators that use a main memory based simulation model (in which different threads are used for executing different agents).

Most systems do not allow defining changes in the environment. For example, if the simulator has only 2D visualizations then it is not possible to add a 3D view to it. They come with their own interpreted language that has a steep learning curve. They are also not powerful enough for expressing common programming constructs. For example, though the interpreted language of NetLogo [5] makes it easier for non-programmers to use the system, the lack of advanced data structures could hamper the use of NetLogo in complex scenarios.

The simulators that employ distributed computing are difficult to set up and maintain. There is no straightforward method of installing and deploying them. The time taken to build and deploy a simulation over a distributed system is considerably high. We have overcome these limitations in our system.

1.1 Our Approach
DMASF (Distributed Multi Agents Simulation Framework) is written in Python [9]. It has been designed to distribute computational tasks over multiple computers and thereby simulate a large number (up to billions) of agents. DMASF is available for general use. In DMASF, an agent is an autonomous entity working towards a goal. A Multi Agent System can be defined as a group of agents working together to achieve their goals.

The DMASF core is small and lightweight. It has been designed such that a user can develop simulations and applications in it without any difficulty and an experienced developer can extend the core and build domain specific applications. The DMASF GUI currently provides a 2D representation of the world. It can be directly extended by the developer to create 3D visualizations of the simulation.

DMASF uses a database for storing agent state information. This helps DMASF in scaling to a large number of agents. It can also be run without the GUI for simulations in which speed is the paramount criterion, animation of agent behavior is not important and cumulative results are sufficient.

In this paper we will discuss the design of DMASF as well as some of the results we have obtained by running various test simulations to illustrate the scalability of DMASF. We will also describe some applications scenarios where DMASF has been used.

1.2 Motivation and Design Goals
The traditional paradigm for Multi Agent Systems is to run each agent in a separate thread. This clearly represents the agents being concurrent autonomous entities. However, this mapping of the agent world to the computer world does not allow for execution of a large number of agents. On a standard desktop computer system (2GHz Processor, 512 MB main memory) we cannot go beyond 1000 threads.

Therefore, we need a different model. The model of execution used by DMASF is similar to the model used by current Operating Systems to run multiple tasks at the same time [10]. Each task is given a time-slice of CPU time. It is then suspended and another task is run. We schedule and execute agents in a similar manner. Each agent has an update function that specifies what the agent does. The update functions for all agents are run over and over in an iterative manner till the simulation ends.

Distributed computing by default is much cheaper than the alternative of doing all the computation on one very powerful machine. DMASF employs distributed computing to obtain high scalability. To keep the hardware requirements of the individual computer systems low, we have designed and built DMASF to have a very small footprint.

This scale up or ability to simulate a very large number of agents helps in increasing the number of domains and scope of application of Multi Agent System simulation techniques. It also enables more fine grained simulation. For example, in a RoboRescue [12] environment it is feasible to simulate 100,000 civilians and their behavior using DMASF.

In a Multi Agent Simulation, especially one that is distributed, it is imperative that the simulator provide the ability to pause and resume the simulation because the required computers may not be available all the time. DMASF allows the user to pause and resume the simulation after each iteration.

Different Multi Agent Systems can have different ways of visualizing the world. Since we did not wish to constrain the

\[1\] The primary authors of this paper are students. The ideas presented in this paper can be demonstrated at the conference.
The developer can not only define what his agents do, but what they look like as well.

The contribution of this paper is in to build a new system with the following design goals:

- A fast lightweight core.
- Distributed computing for high performance.
- Scalability for hundreds of millions of agents without the GUI.
- A separate extensible GUI.
- Saving the simulation state so that it could be resumed at any time.
- Easy extensibility.

In section 2 we discuss the various design issues and challenges faced while developing DMASF. In section 3 we describe how to use DMASF. In section 4 we present the architecture of DMASF followed by Experimental results in Section 5. Related Work is discussed in section 6.

2. DESIGN ISSUES OF DMASF

An agent in DMASF is represented by an agent type and an agent id. Each agent type can have an independent set of properties. Multiple agent types and multiple agents of a single type are permitted.

An agent cannot change its type directly. To implement type changing, the environment would need to kill this agent and create an agent of the new type. In DMASF an agent lives until it is explicitly killed by the environment or the simulation ends.

A host is an instance of DMASF running on a computer. A computer can run multiple hosts at the same time. Our model has a set of simulator objects on each host (refer section 4.1.3). We use a database for storing agent state information. Databases already have excellent query mechanisms and are very robust. Databases can easily store and retrieve information for billions of tuples and thus help in achieving impressive scale ups (refer section 4.1.5). To illustrate this, let us consider 100 million agents. If we use a single integer to represent each agent, it will require 380 Megabytes. This cannot be kept on a computer with 256 Megabytes of main memory. In DMASF, a fixed number of agents are kept in main memory at a time while the rest are flushed to a database system so that retrieval and update is performed efficiently.

Also, we need to provide the same view of the world at each iteration to all the agents. We hence cannot commit the updates made by an agent to the database immediately as the simulation would then present two world views. One has to wait until all agents have finished updating. Again, due to the sheer number of such updates we cannot store these updates in main memory. We thus keep a fixed number of updates in main memory and write the other updates to secondary storage. DMASF has a setting to override this default behavior if the simulation requires updates to be visible immediately (refer section 3.5).

Another challenge in implementing a distributed computational system is to schedule or decide which agents should be simulated on which machines. We cannot allocate an equal number of agents to each host as slower hosts will then tend to slow down the faster ones. Therefore, DMASF has a dynamic scheduler that assesses the performance of each machine and dynamically schedules and load balances agents on them. This scheduler will be described in detail later (see section 4.1.6).

The hosts in DMASF are organized in a client-server architecture. The server decides which agents are to be simulated on which host. It is also responsible for the synchronization among hosts.

Query results that give common world states are cached so that the simulation runs faster. In a simulation in which agents had to move in a circle (refer section 5.2.3), this caching reduced simulation time from $O(n^2)$ to $O(n)$ where $n$ is the number of agents. Whenever an agent requests such information, it is provided from the cache instead of running the query. This reduces the number of database queries, the load on the database system as well as avoids redundant computation.

3. DMAS FRAMEWORK

The main components that define a Multi Agent System are

- (1) Agent Type Definition, (2) Agent Behavior Definition, (3) Agent to Agent Communication, (4) Environmental Definition and, (5) GUI.

3.1 Agent Type Definition

An agent type is defined by a call to the RegisterAgentType function which will tell DMASF the properties and the update function for that type. Each agent type can have a distinct set of properties. Properties can be of the following primitive data types: (1) Integer, (2) Float, (3) String and, (4) Large String. Complex Python data types can be represented using the large string data type and the Python pickle module (which converts any data type to a string). Agents have some default properties such as id (identifier of the agent), size (defines how big the agent would look in the GUI), x, y (specifies the co-ordinates of the agent in the world).

Example: A world with three types of agents - helicopter, smoke and rescue vehicles. The environment would spawn smoke agents randomly representing new fires in the area. The helicopter agents can see the entire world and are responsible for informing the rescue vehicles about the various smoke agents. Here, the rescue vehicle agents would have properties representing which smoke agent it is going to tackle. The smoke agents would have a property that defines how many more iterations until the smoke agent dies (i.e. the fire burns out on its own).

3.2 Agent Behavior Definition

The agent behavior is defined using a class. The user needs to extend the default-agent-handler class that is provided with DMASF. The class already has some built-in functions and properties such as:

- *writeState*: writes the modified properties to the data store.
writeStateSynchronous: writes the modified properties to the data store immediately. These changes are now visible to the next agent in the world view.

sendMessage: to send a message to another agent.

getMessages: to retrieve the messages that have been sent to the agent.

kill: to kill the agent.

fields: Python dictionary that provides the values of the properties of the agent.

db: a low level interface to the database for writing custom SQL queries.

Moreover, the user should overload the update function with the agent behavior. This is the code that is executed by DMASF whenever the agent is scheduled.

We have also provided some simple aggregate functions to help executing common database queries. These can be used for implementing various percepts of the agent. In case even finer control is required, the developer can write SQL queries to directly access the database.

Example (continued): The rescue vehicle agents would contain code that moves the agent in the world towards the smoke agents as well as code for extinguishing the fire. The helicopter agents would contain SQL queries for scanning the world for new smoke agents and then informing the rescue vehicle agents about them. The smoke agents do nothing other than waiting to be either extinguished by the rescue vehicles or die out on their own.

3.3 Agent to Agent Communication

Messages to the agent are represented by a from address (specifying the type and identifier of the agent who sent the message), a simtime (the iteration number at which the message was sent) and the message (a string specifying the contents of the message). Agents can access these messages using the getMessages function. They can send messages using the sendMessage function. There is no limit on the number of messages an agent can receive. By default, messages are flushed after each iteration. This is done to improve the performance. Messages are stored in the database and transferred to the host that is executing the agent code over TCP/IP.

Example (continued): The communication between the agents happens when the helicopter agents inform the rescue vehicle agents about new smoke agents (fires).

3.4 Environmental Definition

Just as the agent behavior is modeled by a class, the environment behavior is also modeled by a class. The user needs to extend the default-world-handler class. The user can overload the following functions:

begin: This function is called at the beginning of the simulation. The user can setup the environment here. Typically, this function will contain code for setting up the world as well as creating the initial agents.

beginUpdate: This function is called at the beginning of each iteration. The agent code is executed only after the execution of this function finishes.

endUpdate: This function is called at the end of each iteration. An iteration is assumed to have completed only after this function finishes execution. If this function returns True then DMASF assumes that the simulation’s goal has been reached. If it returns False the simulation will continue to run.

disableFlushingMessages: By default DMASF will flush messages (clear inbox) for each agent after the end of each iteration. However, if a simulation requires messages to be stored then this function can be used to override the default behavior.

Example (continued): In this simulation, the maximum simtime was set to -1 (infinity) so that the simulation would go on until explicitly killed. Flushing of messages was enabled because we did not need to store messages. Synchronous writes were also disabled because during an iteration, we wished to present the same world view to all the agents.

3.5 Framework wide options

DMASF has some built-in functions to enhance the scope of the simulation as well as tweaking system wide properties.

setMaxSimTime: This specifies the number of iterations for which the simulation should run. By default the simulation will run for infinite number of iterations or until the endUpdate function of the world handler returns True.

disableFlushingMessages: By default DMASF will flush messages (clear inbox) for each agent after the end of each iteration. However, if a simulation requires messages to be stored then this function can be used to override the default behavior.

end: This function is called when the simulation ends. The simulation ends when it has run for the specified number of iterations or by exiting DMASF or the endUpdate function returns True.

3.6 Graphical Interface

The GUI has been developed in OpenGL. The user can write OpenGL code specifying how the agent should be drawn.
While defining the type of the agent, the user can provide a render function that will be called whenever the GUI needs to draw an agent of that type. The GUI can also be configured to draw only specific agents. This is useful for observing the behavior of only a few specific agents.

Example (continued): We provided DMASF with custom render functions to draw the three types of agents. Rescue Vehicles were drawn as a convex polygon in the shape of a car. Helicopter agents were drawn in the same fashion. Smoke agents were drawn by four overlapping circles. Figure 2 shows a screenshot of the GUI.

3.7 Developing and Deploying Simulations
Developing a simple simulation consists of first defining the agent type. Next, the user needs to define the behavior of that type. Once the simulation has been developed it needs to be run on different computer systems. Adding a computer system consists of running the apiclient on that system and telling DMASF to use that particular system as a host. To run the apiclient, the user needs to start the daemon. The user code will automatically be transferred across the network and will be ready for execution.

3.8 Hardware Requirements
DMASF does not require any special hardware to run. We have tested DMASF on machines with 256MB RAM and 1.7 GHz AMD processors. It has also been tested on a computer system with four 3GHz processors and 2GB of main memory. There are settings to allow the user to control the number of agents that should be kept in main memory thereby controlling the amount of memory used by DMASF.

Example (continued): This simulation was run on a single host with four 3GHz processors and 2GB of main memory. Without the GUI, we were able to simulate 1000 rescue vehicle agents, 50 smoke agents and 2 helicopter agents in approximately 5 seconds. With the GUI we were able to simulate 100 rescue vehicle agents, 10 smoke agents and 2 helicopter agents in 5 seconds.

4. ARCHITECTURE OF DMASF
DMASF is written in Python [9] which enables it to take advantage of the Object Oriented nature of Python. The core modules of DMASF are as follows: (1) User Interface, (2) Server, (3) Host, (4) Host Manager, (5) Data Storage, (6) GUI and, (7) Scheduler.

4.1 Components of DMASF
4.1.1 User Interface
The User Interface consists of the functionality required for instantiating and executing the user code. It also manages different agent types. It consists of the agent handlers for each type and the world handler. It communicates with the server.

4.1.2 Server
The server is responsible for synchronizing the various hosts. It is also responsible for setting up the simulation environment as well as transferring the user code to the hosts. The server is responsible for only controlling the simulation. No user code is executed by the server.

4.1.3 Host
A host refers to an instance of DMASF running on a computer system. Each host receives commands from the server which it executes. The commands are: (1) Simulate: this specifies the set of agents this host should simulate. (2) Update database: this tells the host to start committing changed agent states to the database. (3) Simulation Environment Changed: this tells the host that there has been a change such as an agent being created or killed. Each host has a set of simulator objects. These simulator objects run on different threads and are actually responsible for executing the user code for the agents.

4.1.4 Host Manager
The host manager is responsible for each and every host that is connected to the server. Communication is done using standard BSD sockets over TCP/IP. The host manager is also responsible for sending commands to the hosts. It ensures reliable communication by waiting for an acknowledgement for each command.

4.1.5 Data Storage
We use MySQL as the database. The data storage layer provides functionality required for reading and writing agent states to the database. It also provides support for storing and retrieving the messages. It is also responsible for writing agent data temporarily to secondary storage so that excessive amounts of main memory is not consumed.

Databases were used to store agent states by Kota et al [4]. The advantages of using a database are manifold. Databases are built to handle large amount of data. This is extremely useful when we need to simulate millions of agents. Using a database, looking up individual agents is extremely fast. Also, percepts can be expressed directly by giving SQL queries.

4.1.6 Scheduler
The scheduler runs on the server. It is responsible for deciding what each host should simulate. The basic algorithm takes into account how long each host took to process the set of agents that it was given for the last iteration. It then attempts to come up with a dynamic scheduling policy.

The simulation can be defined as the execution of the agent code for all agents of all types. We assume that agents of the same type require approximately the same time to get processed in the same iteration on the same host for the purpose of scheduling. This assumption is justified because agents of a similar type are expected to perform similar functions.

The scheduler has to schedule the simulation over all the hosts. The best results can be obtained when the hosts are not idle and are performing parallel computation. Thus the objective of the scheduler is to minimize the idle time (the time when a particular host is idle, while the other hosts are still computing) for all hosts.

DMASF initializes by distributing equal number of agents to each host for all types i. Let T be the run time of a particular iteration of the simulation. Thus, T depends on the maximum run time among all the individual hosts, because the slowest host will define the run time of that iteration. The objective of the scheduler is to minimize the run time of the simulation, or to minimize T. We
do this by identifying the slowest host for each iteration and then allotting it the least number of agents to simulate. Similarly, the fastest host is allotted the maximum number of agents to simulate.

The number of agents of one type allotted to one host is inversely proportional to the time it takes to simulate one particular agent of that particular type. It is inversely proportional because the faster host should be given more agents for simulation. Dividing the agents in such a ratio ensures that all the hosts finish processing their sets of agents at approximately the same time, thereby reducing the idle time.

DMASF also includes a static scheduler that schedules equal number of agents of each type on each host. The user can configure DMASF to use either of the two schedulers.

### 4.1.6.1 The Two Schedulers Compared

We ran the circle simulation (described in detail in section 5.2.3) separately using each of the schedulers. The simulation was run for one hundred thousand agents on two hosts. The first system (Machine A) had four 3GHz processors and 2GB main memory. Machine B had an AMD 3000+ (1.6GHz) processor with 512 MB main memory. Table 1 and Table 2 below show the number of agents simulated by each machine and the time required per iteration.

#### Table 1. Static Scheduler

<table>
<thead>
<tr>
<th>Iteration Number</th>
<th>Machine A</th>
<th>Machine B</th>
<th>Total Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of Agents</td>
<td>Time Taken (secs)</td>
<td>Number of Agents</td>
</tr>
<tr>
<td>1</td>
<td>50000</td>
<td>19.64</td>
<td>50000</td>
</tr>
<tr>
<td>2</td>
<td>50000</td>
<td>19.68</td>
<td>50000</td>
</tr>
<tr>
<td>3</td>
<td>50000</td>
<td>19.55</td>
<td>50000</td>
</tr>
<tr>
<td>4</td>
<td>50000</td>
<td>19.59</td>
<td>50000</td>
</tr>
<tr>
<td>5</td>
<td>50000</td>
<td>19.93</td>
<td>50000</td>
</tr>
</tbody>
</table>

#### Table 2. Dynamic Scheduler

<table>
<thead>
<tr>
<th>Iteration Number</th>
<th>Machine A</th>
<th>Machine B</th>
<th>Total Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of Agents</td>
<td>Time Taken (secs)</td>
<td>Number of Agents</td>
</tr>
<tr>
<td>1</td>
<td>50000</td>
<td>19.68</td>
<td>50000</td>
</tr>
<tr>
<td>2</td>
<td>80766</td>
<td>31.34</td>
<td>19234</td>
</tr>
<tr>
<td>3</td>
<td>76551</td>
<td>30.08</td>
<td>23449</td>
</tr>
<tr>
<td>4</td>
<td>73473</td>
<td>28.83</td>
<td>26527</td>
</tr>
<tr>
<td>5</td>
<td>73860</td>
<td>28.95</td>
<td>26140</td>
</tr>
</tbody>
</table>

This shows that the dynamic scheduler gives a significant performance boost if the hosts have different computational capabilities.

The simulation was then repeated and run on three hosts. Machines A and B were the same as the ones used in the previous experiment. We added another machine, Machine C with an Intel Celeron Processor (2.6 GHz) and 512MB main memory. The allocation of agents by the schedulers is shown in Figure 1.

### 4.2 Applications

We designed and built several simulations using DMASF. The first application that was built was a model of the Wumpus Environment [11]. The Wumpus environment is a grid based environment with squares containing gold as well as obstacles. An agent starts at a random position and must find the gold while avoiding the obstacles.

DMASF was also used to build the infrastructure for a car racing competition. The environment consisted of a track and the agents were the various race cars. The race cars could ask the track about the next turn as well as about the location of the other agents.

We further built a FireFighter simulation modeled along the lines of the RoboRescue [12] competition. It had three kinds of agents: Helicopter, Rescue Vehicle and Smoke (refer to Figure 2). The environment would randomly spawn smoke agents. The helicopter would be scanning the world looking for new smokes. On finding a new smoke (fire), it would alert the rescue vehicle agents. Each rescue vehicle agent would go towards the smoke closest to it and try extinguishing the fire. The smoke agents would have a natural lifetime that corresponds to the time in which the fire would burn out on its own.

![Figure 1. Agent allocation by schedulers](image1.png)

![Figure 2. FireFighter simulation](image2.png)
5. EMPIRICAL VALIDATION

5.1 Experimental Setup
The experiments were conducted on a machine with four 3GHz processors and 2GB of main memory. The MySQL version used was 5.0.18. The Python version used was Python-2.3.

5.2 Sample Worlds and Agents

5.2.1 K-P Simulation
One of the simulations created was the K-P benchmark, where K refers to the number of agents in a group. P is the number of messages sent by each agent on receiving P messages from the other agents in the group. This simulation was designed to test the efficiency of messaging.

5.2.2 Crawler Simulation
We used the “crawler” simulation as another benchmark. It had two types of agents: worker and scheduler. The scheduler assigns tasks to the worker agents by sending the worker agent a message. These worker agents then perform this task (modeled by waiting for a certain amount of time) and send the results back to the scheduler agent. We used two types of models. One model had worker agents using the system call sleep (to simulate a task such as waiting on a socket or writing to files) and the other had a busy wait. Both the models used a wait time of 10ms per agent. This simulation was designed to test the way agents were executed on each host. If an agent is waiting on a system call, the entire simulation should not wait; another agent should be simulated.

5.2.3 Circle Simulation
The circle simulation environment is where the agents are spawned in a random manner in a 2D plane. This is an extension of one of the examples provided with NetLogo [5]. The agents can only see other agents. The agents must move around in a circle whose center depends on the location of the other agents. This simulation was run on two different setups. The first setup involved two hosts running on a machine with four 3GHz processors and 2GB of main memory. We gradually increased the number of agents to observe how well the simulation would scale. The simulation scaled linearly (refer section 5.3.3). In both the setups the MySQL database was on this machine.

The second setup involved running the simulation with one million agents. We increased the number of hosts and plotted the resultant gain in performance (refer to Figure 6). The machines that were used for simulation were standard desktop computers with AMD 1.6GHz processors and 512 MB of main memory.

5.2.4 RTT Simulations
We measured the RTT (round trip time) of each message sent between a sender and a receiver agent [2]. This involved putting a timestamp on the message when it was sent and then subtracting this from the current time when it was echoed back by the receiver. For each pair of sender and receiver, we calculated the RTT from an average of 50 messages. We also measured what happens when one particular agent is flooded with a large number of messages. For this we created one receiver agent and many sender agents and observed the values of the RTT. The results are explained in section 5.3.4.

5.3 Results

5.3.1 K-P Simulation
The simulation was run with 10000 agents and two hosts (Tables 3 and 4). All times are in seconds.

Table 3. K/P Simulations for small values of K

<table>
<thead>
<tr>
<th>K/P→</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>9</th>
<th>19</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.52</td>
<td>2.83</td>
<td>3.77</td>
<td>4.45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>2.47</td>
<td>2.67</td>
<td>3.07</td>
<td>3.56</td>
<td>4.58</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>2.43</td>
<td>2.53</td>
<td>2.67</td>
<td>2.87</td>
<td>3.35</td>
<td>4.77</td>
</tr>
</tbody>
</table>

Table 4. K/P Simulations for large values of K

<table>
<thead>
<tr>
<th>K/P→</th>
<th>1</th>
<th>19</th>
<th>49</th>
<th>99</th>
<th>499</th>
<th>999</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2.48</td>
<td>2.91</td>
<td>3.64</td>
<td>4.58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>2.20</td>
<td>2.53</td>
<td>2.69</td>
<td>2.94</td>
<td>4.83</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>2.13</td>
<td>2.39</td>
<td>2.51</td>
<td>2.63</td>
<td>3.63</td>
<td>4.63</td>
</tr>
</tbody>
</table>

The graph of Time taken v/s P (the number of messages sent in a group) is shown in Figure 3.

Figure 3. K/P Simulation

Since messages are being stored as tuples in a table, we expected a linear increase in the time with a linear increase in the number of messages in the system. It can be observed that as P increased, the number of messages in an agent group increases. Therefore with linear increase in the number of messages the simulation time also increased linearly. For the same P, with a larger K (group size) there are fewer total messages in the system and hence the time taken is less.

5.3.2 Crawler Simulation
We expected a linear increase in the total time taken for each iteration. We were able to achieve this linear increase (as shown in Figure 4). We expected the simulation to take less time with the sleep system call as we have a number of simulators executing the agent code in separate threads. Thus, if an agent executes a system call in a simulator, another simulator would get scheduled to run. The time difference between busy waiting and using the sleep system call is clearly visible (in Figure 4) with the former taking more than two times the latter. In the busy waiting case each agent actually waited for 10 milliseconds.
5.3.3 Circle Simulation

We got a linear increase in the time taken per iteration (as shown in Figure 5) as we increased the number of agents. We varied the number of agents from 100 to 100 million. By extrapolating the linear results that can be seen in Figure 5, we can simulate one billion agents in approximately 250,000 seconds (70 hours).

5.3.4 RTT Simulations

As expected, we got a linear increase in the RTT values (as in Figure 7). This is because as the number of agents increases, the intervals at which they were scheduled to run also increases.

We further got a linear increase in the RTT values when one agent was flooded with messages. As we increased the number of sender agents, the simulation time also increased linearly (as depicted by Figure 8). This implies that the DMASF messaging subsystem was able to handle both the case of a large number of messages in the system as well as the case of flooding (i.e. a lot of messages being sent to a single agent).
6. RELATED WORK
A lot of work has been done in the field of developing Multi Agent System Simulators. SPADES [1], MASON [3], NetLogo [5] and JADE [7] are some of the more popular simulation toolkits. JADE (Java Agent Development Environment) which is a middleware for developing and deploying Multi Agent System Applications. SPADES is also a Distributed Multi Agent Simulation Environment which is not language specific and allows agents to be written in any programming language. The agent code interacts with SPADES over UNIX pipes. MASON is a light, fast, scalable discrete-event Multi Agent simulation library written in Java. It also has a separate visualization layer for viewing simulations. NetLogo is a desktop simulation toolkit that scales well for small number of agents. It uses its own interpreted language for writing agent simulations. FIPA [6] is the standard for Multi Agent System Simulators. DMASF is partially compliant with FIPA standards. We are in the process of making DMASF fully FIPA compliant.

7. CONCLUSION
Multi Agent System technology can be used to simulate complex environments at both microscopic and macroscopic levels. Therefore, it is required to have a simulation toolkit to cater to both these needs. Many of the current Multi Agent Simulation toolkits either cater to microscopic simulation for very small environments (~10,000 agents) or do only macroscopic simulations.

One of the challenges taken up in this paper is to provide a generic toolkit that can simulate a very large number of agents in a relatively short time by using distributed computing. Our results show that we can potentially simulate one billion agents in around 250,000 seconds. With increased number of systems used for simulation it is feasible to bring down this time further.

One major advantage of our simulation toolkit is that it can be used to rapidly implement and deploy small Multi Agent System driven simulations. Thus, it is amenable for use in Multi Agent System course projects.

Since DMASF is based on Python it is easy to plug-in existing MAS decision modeling systems such as MDP into DMASF.

One of the limitations of our system is the computational mismatch between the GUI to show the simulation results and the backend that actually does the simulation. With advances in GUI rendering and related technologies it would be feasible to have real-time observation or animation of a very complex Multi Agent Simulation with one billion agents. If such GUI technology is not available then other appropriate solutions need to be found for handling this mismatch. We are currently working on simulation of very complex environments using this toolkit.

8. REFERENCES