Final Report

CMPT 431 - Spring 2013

Group Team Team
Michael Lim - mjlim@sfu.ca
Karl Olson - karlo@sfu.ca
Cody Hart - chart@sfu.ca
Tim McKenney - tjm1@sfu.ca
Jared Daley - jsd12@sfu.ca

Project Overview

For our group's solution to the n-body interaction simulation, we have developed a multi-threaded, multi-hosted system that is able to comfortably accommodate one million separate bodies and has been verified to be correct. The system architecture is of a modified client-server paradigm with peer-to-peer communication between clients to facilitate the movement and detection of bodies between nodes.

The system has been developed using two programming languages: Python and C++. While the client was already implemented in C++ at the beginning of the project, the group decided that the server and tool set that needed to be developed should be done so in Python as it is a relatively easy language to use, especially next to C++, and making the client and servers interact via sockets would be rather trivial. Developing the client further in Python was also looked at, but initial research showed that learning the C++-Python API would be difficult to learn in the time we were allotted for the project. In order to facilitate the communication between nodes, we created our own application layer protocol that facilitated all requests between client, server, and visualization without error.

While running, the main server is responsible for the initialization of the system and the distribution of work among the available clients, as well as the synchronization of updates between the clients and visualization, when used. The main server is also responsible for the distribution of information about clients to other clients, i.e. which client owns a particular cell and the IP address of that client.

The clients are responsible for handling their own set of bodies, enacting on them in a particular fashion when the server tells them to. These server-controlled actions include updating the sensors of each of the bodies, updating the position of each body, and getting every cell a copy of the positions in their population after a position update. The clients also respond to other clients requesting the positions of bodies in adjacent cells and sending population data to the visualization client when there is one on the network.

The visualization client is responsible for rendering visual output requested from or pushed from the other clients. Initially, the visualization client displays a heat map created by converting the value of the population in a particular cell to a corresponding color. However, individual bodies can be displayed in several different ways if needed.

The system was developed in several stages, the first of which being to take the initial brute force implementation of the main program and improving performance and efficiency. We were to do this by using smartly-chosen data structures and having each robot look at only one subset of the overall population, rather than all robots. Rather than having one large n^2 problem, this improved performance by splitting it up into several smaller n^2 problems.

The next stage was to multi-thread the system so that several iterations of each n^2 problem could be run concurrently. To do this, we used POSIX threading and made each major step of the program, sensor update, position update, boundary traversal, and population propagation, sync to preserve correctness.

The final stage of development was for this to work as a distributed multi-system environment so several iterations of the multi-threaded system could be run concurrently. As stated, this was done using a custom application layer protocol and more POSIX threading to handle several inbound and outbound messages at a time. A server was implemented in order to facilitate propagation of information regarding each client and the syncing of all clients across the distributed system.

Implementation

Before an in-depth discussion regarding the implementation of our solution, it is important that you familiarize yourself with the different files used in the implementation, as well as understanding the protocol developed to facilitate inter-client and client-server communication.

I. Files and Their Purpose

(1) **Universe** (the client program)

1 **universe.cc** - The client itself. This part of the program is responsible for the main functionality of each client. It performs the distance calculations between robots in the universe, moves bodies into position, maintains the body population, and facilitates communication in and out of each client node.

2 **universe.h** - The universe header file.

3 **controller.cc** - Gives the bodies their linear and rotational speeds.

(2) **uniserver.py** - The main server. Responsible for communicating to the clients which portion of the world they're responsible for, synchronization of the universes to ensure correctness, and the distribution of contact information (IP addresses) for clients in the case that inter-client communication needs to occur.

(3) **display.py** - The visualization client. Connects to the main server to stay in sync with the universes and connects to the universes to gather information about cell population and body position so it can display a visual representation of the universe at any given step in the system.

(4) **Tool Set**

1. **startupServer.py** - Server program that can send terminal commands to a set of clients connected on other machines using startupClient.py. This makes it possible to start a large number of universe clients with one centralized command.
2. **startupClient.py** - Connects to the startup server. Once a client has been connected, the startup server can control all of these clients with a single terminal command rather than having to copy and paste each command on each machine.

II. Application-Layer Protocol Commands

Please note that all messages sent across the network are of the form <L1><L2><Message/Command>. L1 is an integer from 0-9 denoting the length of L2. L2 is an integer of *length* 0-9 denoting the length of the sent message or command. The final part is, of course, the message or command sent between the two nodes of the network.

(1) "**SUP.**" - When a client connects to a server, this command initializes communication and cell assignment.

(2) "**id uni**" - This command is sent from a client to the server telling the server that it identifies as a universe (a main client).

(3) "**id vis**" - This command is sent from a visualization client to the server telling the server that it identifies as such.

(4) "**+100 <message>**" - The server sends out this message denoting that identification has succeeded.

(5) "**-<number> <message>**" - The server sends out this message denoting that a non-recognized command has been entered.

(6) "**cells**" - A client asking the main server which cells (portion of the world) it is responsible for.

(7) "**CELLSASSIGNED**" - The server broadcasts this message to each connected client to let them know that all cells have been assigned to clients.

(8) "**whohas x y**" - This is sent from a client to the server when a client needs to know which client is the 'owner' of a certain cell. x and y are the Cartesian coordinates of a cell across the entire universe.

(8) "**READY**" - The clients broadcast this message when they finish their steps before the program can begin. In the case of the main client, when they finish allocating cells. In the case of the visualization client, pairing the list of cells to a list of IP addresses.

(8) "**GRA**", "**UPE**", "**USN**", "**UBT**" - The main server communicates one of these to all the clients telling it to perform the denoted step after every client has completed the previous one (see below command). In each respective case, grabbing robot array for each cell (thread), updating body position for all bodies in that node's population, updating body sensor for all bodies in that node's population, and boundary traversals between two remote clients.

(9) “**DGR**”, “**DUP**”, “**DUS**”, “**DBT**” - The client communicates one of these to the main server when it is done a particular step described in (8) of this section.

(10) “**HUH x y**” - An inter-client command asking for the contents of a particular cell. x and y are the Cartesian coordinates of a cell. This can also be sent from a visualization client in the case that it has been asked to draw the bodies in each cell.

(11) “**NB x y a cx cy**” - An inter-client command telling a client that a new robot is about to enter a portion of the universe they control. 'x' and 'y' are the coordinates of the robot and 'a' its angle. 'cx' and 'cy' are the coordinates of the cell this robot should be placed inside.

(12) “**CELLPOPALL**” - A command sent from the visualization client to a main client asking a universe the to broadcast the population of each cell it owns at every step so it can draw it.

(13) “**BYE**” - A client telling another client it is closing its socket connection.

(14) “**KILL YOURSELF**” - A command broadcast from the main server telling all connected clients to exit due to an error occurring across the system.

III. System Functionality Overview

Our program is made up of three distinct parts. The main clients run on various hosts across a network and are responsible for the management and movement of bodies between cells on a given host and between cells on two distinct hosts. There is also a main server that provides synchronization for all the clients, the propagation of contact information (IP addresses) to facilitate inter-client communication, and the assignment of a contiguous set of cells to the attached clients. Optionally, there is a visualization program that is responsible for rendering the state of a system visually using the population of each cell, as well as the position of bodies, across the entire system. The main server and visualization are not processor-intensive programs and are bottlenecked by only the network, so these two processes can be run on one host if needed.

IV. Main Client

The main client, or several instances of it, is the workhorse of the entire system. It is written in C++ and is based on the initial version of the universe program given to us at the beginning of term.

Upon startup, the client sets up according to the arguments passed to it. It then partitions the world into a grid based on the variable 'matrix_size'. It then opens a socket with the main server and initiates a handshake with the main server. The client tells the server that it is a client using the 'id uni' command and, upon a response from the server, requests the cells it is responsible for. Once it has received the cells in the form of a large string of characters, it parses this string and sets up a 2D-array gridset[x][y] with a Boolean value whether x,y is owned by this client or not. It then starts a thread to listen for incoming connections from other clients.

A thread is started for each cell owned by the system. This means that each cell can act (theoretically) concurrently. A separate population for remote robots is initialized at this point, in order to keep them separate from the owned population. An array to keep information regarding the address of other clients is also initialized here. The client tells the server it is ready to begin and initialization of the client is finished. The server will then broadcast the first step to all clients on the network.

Before the first step occurs, the population of bodies is initialized. It is important to understand that each client initializes the entire population, even in cells that it doesn't own. Every client has its own copy of the entire population and world. Even though it doesn't act the entire thing, it is important to have an entire system's worth of resources at the client's disposal in the case of edge cases. The client uses only the bodies that it creates on startup, so if the entire population ends up in only its set of cells, the client would require that entire population's worth of bodies.

Once the population is initialized, it is put in to the main loop. The first command it receives from the server is 'GRA', meaning that each cell (thread) should make a copy of the current population of its cell. This allows us to let each cell act on its own set of bodies without having to lock a global data structure during the upcoming processingheavy sections of the program and ruining the multi-threading effect we are trying to achieve. Once this is done, it sends "DGR" to the server and waits for the next step to be received.

The next step is 'UPE', which stands for 'Update Pose'. This step is meant to update the position of each of the bodies based on the bodies nearest it vis-a-vis collision avoidance. The first part of the step is to handle bodies that are new to the system, i.e. has moved from a remote client to this one. It then changes its position on the plane according to its speed and current position. Finally, if has traversed the boundary of a cell, it removes itself from that cell's population and puts it in its new one. Once done, it sends 'DUP' to the server and waits for the next step.

While we've been doing the position updates, each client has been sending and receiving "NB x y p" messages. 'NB' stands for 'New Body' and signifies that a body has crossed the boundary between two clients. Therefore, one client sends the other an 'NB' message to tell the former that a new body has appeared in its territory. x and y are the Cartesian coordinates of the body across the universe, and p is its angle. x, y, and p are denoted in hexadecimal float format. The reason for this is because it provides the greatest accuracy with the smallest amount of overhead. Each client takes in these NB messages and queues them for processing in the next step.

The next step is 'UBT', which stands for 'Update Boundary Traversals'. It takes the queued set of NB messages from the previous step and adds the new bodies to the population. It reuses non-used bodies initialized at the start of the client in order to do this. We did this to save on memory and removal from hard-to-mutate data structures. With the new bodies in place, we need to get rid of the bodies that left the system as well and mark them as unused. Having finished this, the client sends 'DBT' to the server and waits for the next step.

Next is 'USN', which stands for 'Update Sensor'. This is the last step in the main loop and is meant to update each body's 'sensor', or its detection system for bodies that are close to it. It uses this sensor information to determine its position in the next step. This is arguably the most complex part of the program as we sometimes have to take into account bodies running on other clients in remote hosts. For each body, it clears the body's sensor information from last step. It then finds a bounding box for the body based only on its range, and finds all the cells that intersect this bounding box and places them in an empty vector.

If the current client 'owns' the cells in the set of cells that need to be checked for adjacent bodies, then we have the information at hand on the system. However, if a cell isn't owned by the current client, the population of robots in that cell must be sent by the remote 'owning' client for processing. We do this using the QueryRemoteCell function.

The QueryRemoteCell function, if it doesn't have it, asks the main server for the IP address of the host running the client that owns the cell in question. It then either uses an open connection with the remote client, or opens a new one and stores the socket in a list of open connections. Over this new connection, it then sends the "HUH x y" command and is responded to with a list of x,y,p tuples (see above for explanation of x, y, and p). Each tuple represents a body in the cell in question. It then puts these remote bodies into a 'fake' cell, and puts that into the list of remote cells.

Having acquired all the remote data it needs, it then proceeds with the updating of its sensor. Looping through all the robots within the range of its sensor, given at startup. It then is able to determine the closest body to it, whose position will affect its position in the next iteration of the main loop. Having finished, it sends back the 'DUS' command to the server and waits for the 'GRA' command in response, starting the loop over again.

<u>V. The Main Server</u>

The main server is responsible for system-wide bookkeeping and synchronization. We wrote it in Python as it is generally an easier language to deal with, and because speed was not a factor, Python was the perfect candidate.

Upon startup, it parses arguments and assigns variables based on those arguments. It then listens on port 12345 for incoming connections from a certain number of clients. The number of main clients and visualization clients are denoted separately. For every client that connects, it sends out a 'SUP.' message, telling the client to go ahead with its initialization 'handshake'. For every identification, it increments a counter of main and visualization clients. Once the correct number clients have connected, it then assigns a contiguous set of cells to each main client. It does this in one of two ways. If the number of clients is a square (2, 4, 9, 16, etc.), unless told otherwise in the arguments, it assigns cells in groupings of squares. Otherwise, it assigns them in strips.

It then starts a thread devoted to each connected client. While the socket is open, it processes each of the commands sent to it by each of the clients. If a client needs an IP address for a remote client, the client asks 'whohas x y' where x and y are the Cartesian coordinates of a cell across the whole universe.

It also sends out the synchronization messages. It broadcasts the 'GRA', 'UPE', 'UBT', and 'USN' messages, telling each client to perform a particular step. When it receives the appropriate responding 'I'm Finished' messages from each of the main clients, it proceeds to the next step. It is this mechanism that allows us to ensure correctness across the whole system. Finally, if a client dies, it determines that an error has occurred in the system and forces all other clients to die as well. It then restarts itself and waits for clients to reconnect to start the simulation over.

VI. Visualization Client

The visualization client is responsible for the visualization of the simulation. It is also programmed in Python and uses the PyGame library to easily deal with the visual aspects of the program. It has two main threads: one for dealing with networking, and one for dealing with graphics.

In the networking thread, it connects with the server and identifies as a visualization. It then waits for the "CELLSASSIGNED" message to be received from the server. This signifies that all the cells have been distributed to the clients and that it can start polling for information to display. For each cell in the grid, it asks the server which client owns it. Once it gets the information it needs, it contributes a "READY" message to the server, telling it that it can proceed. The system will then proceed with running, basically oblivious to the actions of the visualization client.

Having the IP address of the owners of each cell, it then creates one socket with each host. Note: this isn't one socket for each cell, it is one for each host. It then spawns a thread for each host. This thread subscribes to each client using the "CELLPOPALL" command, which gets each client to send the population of each cell on each step. This comes in the form of x y pop, where x and y are the coordinates of the cell, and pop is the population. The thread then parses this information and repeatedly updates the population for each cell.

In the main thread, if body-drawing is enabled for a cell, it waits for body sensors update step (i.e. the 'USN' command to be broadcasted). Once it has, it sends out a set of 'HUH x y' messages to each of the enabled cells to get the position of each body to draw. It stores the response (whose form is shown in the main client section) in a list of bodies to be drawn. It then updates a global structure of bodies to draw using a lock.

In the graphics thread, a loop is immediately started even before any updates from the clients are received. Until the network thread updates the population count for each cell, it will display a test pattern that is automatically generated on instantiation of the visualization client. It will also determine which average cell population is ideal given the total number of matrix squares and the population. If the total population exceeds the number of matrix squares, the total population divides by the total number of matrix squares is used as the average. Should the opposite be true, population is either divided by the root of the total number of matrix squares, or used directly in the case of very small populations that are less than that root.

Once the main client starts to update cell populations, the graphics thread feeds these populations into the colorize function. This function takes the average cell population and the actual population of the cell. By dividing the population by that average and multiplying it by 256, the function converts the cell population into a full color spectrum by using an HSV-based set of cases that decides which of the R, G and B color variables are being altered by the population size in that cell. Colorize then returns an RGB tuple to the main graphics loop which is used to draw the heat map at the given the x, y location. In an evenly distributed scenario, this will essentially result in a histogram map that's basically all one color, though should any flocking behavior occur, the path of the robots will be evident in color movement across the map.

The graphics loop then makes a local copy of the master robot array and proceeds to draws all of robots in it. It draws the robots by first taking the float values provided by the universes and converting them into integer coordinates that can be rendered on screen. Depending on which options are selected, it will either pass these coordinates and the rotation of robot to a function that returns them as vector triangles (similar to the original visual implementation in C++ provided with the universe,) or it will use PyGame's built in rendering to draw robots as a sprite. Having combined the sprites or vector robots on to the histogram map, the final frame is drawn on the screen.

The graphics thread then handles any key and mouse inputs as per PyGame specification. Regarding mouse input, the left click of the mouse can be used to move around the screen, the right click enables/disable retrieval of the list of robots in the cell at that x,y location and the scroll wheel controls the zoom of the map. The keyboard is used in a similar fashion: the ‘z’ key resets the zoom to it’s default setting of displaying the entire world map, the ‘a’ key inverts which cells are querying for robot population and the ‘s’ key switches between drawing vector robots and sprites.

VII. Error Handling

Very little error handling has been implemented into the system in terms of inside each main client. As long as a valid value is received when expected, the program will continue to run without interruption. However, global errors across the network are handled quite extensively. It is this, matched with the synchronization of the main server that allows us to ensure correctness. The main possible errors are as follows:

(1) The main server will wait for an exact number of clients. If this number is not reached, the system will never start.

(2) If a client crashes, the client disconnects from the server. The server, detecting a closed socket, will kill all other clients and restart itself. It will then wait for connections from clients under the same parameters as was set during its last iteration.

VIII. Known Bugs

(1) The system segmentation faulted using a large number of hosts, 25, to run the simulation. This is probably due to network overload, or possibly due to a restarted machine during initialization. Our system has been tested with 16 hosts and has performed well without error.

(2) Parameter disparity between client, server and visualization will usually result in the system crashing pretty quickly. This remains one of the biggest problems with the system. A lot of parameters must be passed to each part of the system, and if one is forgotten, or if one is set wrong, then this incongruence can cause many things to happen, leading to a client or server crashing. Usually, a client will expect a message and doesn't get it, or doesn't expect one and gets one, leading to a bound error of some kind. This inevitably leads to the clients dying and the server restarting itself.

(3) The non-distributed version of the universe does not run without segmentation fault. Therefore, the DISTRIBT flag must be on when compiling.

(4) When running a certain number of steps, the clients complete and finish, but in doing so, disconnects from the server, causing the server to restart.

How to Use

This will describe how to get our simulator up and running. The process will be described here and commands and paths will be shown below.

To run the system, go to the main Universe directory. You must first compile the universe using 'make universe'. Be sure that the flags you want to set in the header file are set. GRAPHICS turns the local graphics on or off. DISTRIBT takes the universe in and out of distributed mode. NETDEBUG displays all the messages sent and received by the clients. Now, the main server must first be started. Generally, you must give it the number of hosts (main and visualization clients), the number of universes (main clients), and the matrix size of the universe. Once it is up and running, it will wait for clients to connect.

Next, run the Start-Up Server, giving it the port you'll listen on for each Start-up Client, as well as the intended number of clients that are connecting. Next, run the Start-Up client on each of the hosts you intend to run the main clients on. This will allow you to send the universe program and run one single command that will propagate to every connected client, saving your fingers from much tedious typing. You can, of course, do all of this without this tool. But you must run the universe simulation with the exact same commands across multiple computers.

Assuming that you're in the directory with the compiled universe file, typing 'send universe' into the Start-up server will transmit the universe program file from your server to the Start-up clients. You can now run the universe simulations (main clients) using the commands below. If you opted to have results outputted (-t), when each client is finished, type 'get output' to get a concatenated file composed of results from each client. If you want to start a visualization, start the python program giving it the IP of the server, the port to communicate with each client on, and the matrix size.

Main Client

```
Command: ./universe <args>

Arguments:

-? : Prints this helpful message.
-b draw bounding box for each robot
-c <int> : sets the number of pixels in the robots' sensor.
-d  Disables drawing the sensor field of view. Speeds things up a bit.
-f <float> : sets the sensor field of view angle in degrees.
-i <string> : Sets the IP address of the server.
-j Enable optimized creation of cell list in the field of view.
-k Don't draw robots.
-m <int> : sets width and height of the matrix
-n Draw gridlines.
-p <int> : set the size of the robot population.
-P <int> : sets the port for the client.
-q : disables chatty status output (quiet mode).
-r <float> : sets the sensor field of view range.
-s <float> : sets the side length of the (square) world.
-t output robot locations to a file
-u <int> : sets the number of updates to run before quitting.
-v draw triangle robots.
-w <int> : sets the initial size of the window, in pixels.
-z <int> : sets the number of milliseconds to sleep between updates.
```

Main Server

```
Command: python server/uniserver.py <args>
Arguments:

-h : Prints Help Message
-n <int> : Specify the number of hosts (main and visualization clients)
-m <int> : Specify the matrix size
-p <int> : Specify the port to listen on
-q : Turns off verbosity
-u <int> : Specify the number of universes (main clients)
```

Visualization Client

```
Command: python uni-vis/display.py <args>

Arguments:
-h : Prints Help Message
-i <string> : Sets the main server's IP address
-m <int> : Specify the matrix size
-o <int> : Specify the population of the world
-p <int> : Sets the server's listening port
-P <int> : Sets the universe's listening port
-s <int> : Sets the world scale
```

Start-Up Server

```
Command: python server/startupServer.py <args>
-h : Prints Help Message
-c <int> : Sets number of clients to connect
-i <string> : Set current IP address
-s <int> : Set Client Port
-S <int> : Set Server Port
```

Start-Up Client

```
Command: python startupClient.py <args>
-h : Prints Help Message
```

```
-i <string> : Sets Start-Up Server IP Address
-P <int> : Sets Start-Up Server Port
```

Commands for Start-up Server

```
send universe : send a copy of universe from the working directory on the local computer to
the client.
kserv : will kill the uniserver if one is currently running on the same physical pc
start default server : If server/uniserver.py is available, starts it up using the default
settings.
start server: If server/uniserver.py is available, starts it up.  Will prompt you for settings.
t: This command is intended for testing and gathering data only.
get <string> <dest>: Will get a copy of whatever file you tell it from all the clients(in the
remote cwd), concatenate it and save a file called <dest> on the local computer.
local: allows you to run a local command in the cwd
resend: allows you to resend the last used command
h: prints help messages
exit: exits the start-up server and all connected start-up clients
```

System Testing and Results

In order to accurately show our system's capabilities, we ran a large number of tests to try to get accurate data as to our system's speed and scalability. We ran a number of tests at varying levels of population, matrix size, world size, field of view and range of view. In addition to varying these argument values, we also reran the tests with 4, 9 and 16 hosts running the universe program. The following chart shows the various values that were used in testing;

Number of Hosts	Populations	Matrix Size	World Size	Field of View	Range of View
4	1000, 200000	20, 40, 60	50, 100, 200	30, 90, 180, 270	0.5, 1, 2
9	1000, 200000	30, 60, 90	50, 100, 200	30, 90, 180, 270	0.5, 1, 2
16	1000, 200000, 1000000	40, 80, 120	50, 100, 200	30, 90, 180, 270	0.5, 1, 2

From these numerous tests, we were able to see in what conditions our system behaves optimally and in what conditions our system does not. Of these many varying arguments, the few that made the largest difference to the overall execution time were matrix size, host count and population. This is as we expected since higher matrix size should mean less comparisons between unnecessary robots and increase execution speed, more hosts allows for more computing power and should result increased execution speed, and a higher population requires more calculations and should result in a slower execution speed. All of these assumptions proved true for the general run case however there were a few interesting results at certain specific levels. It is important to note, that since varying field of view and range of view parameters are being averaged in our data, the average runtimes displayed are longer than in the optimal case.

We can see the general trend of execution speed from figure 1.1. We can see that with a population of only 1000, the cost of adding additional hosts actually negatively impacts the execution speed of the overall system but with a larger 200,000 population, the addition of more hosts greatly improves the execution speed. The tests, under the parameters we were using, for a population of 1,000,000 were too slow and therefore only the 16 host case is included here for reference.

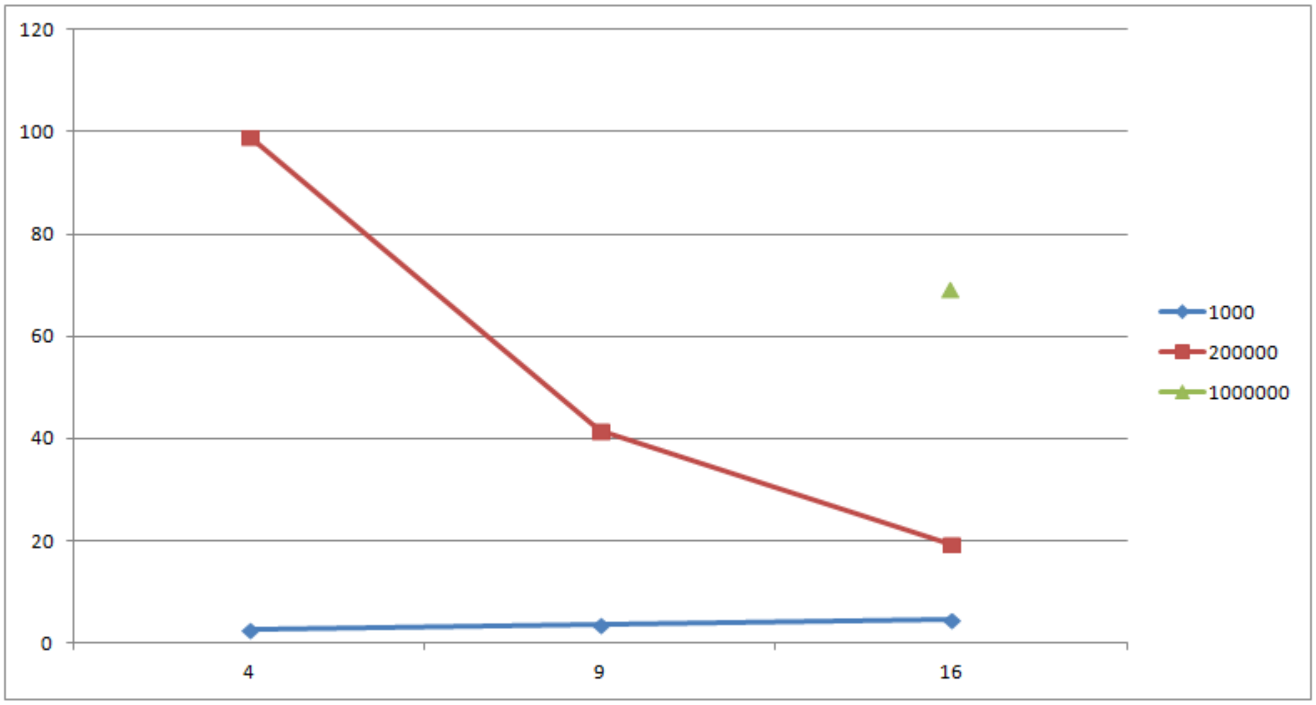

figure 1.1: Fifty Update Average Elapsed Execution Time (sec) vs. Number of Hosts for 1k 200k and 1M Pops

Figure 1.2 helps to show how the varying of matrix size effects the execution speed of the system. From the graph, we can see that for lower populations the varying of the matrix size does not make a very large difference to the overall speed but for a population of 1,000,000 the difference in speed between a matrix size of 40 and 120 is quite dramatic, an improvement from 96.72s to 53.25s. The results for this graph were from running the system on 16 hosts.

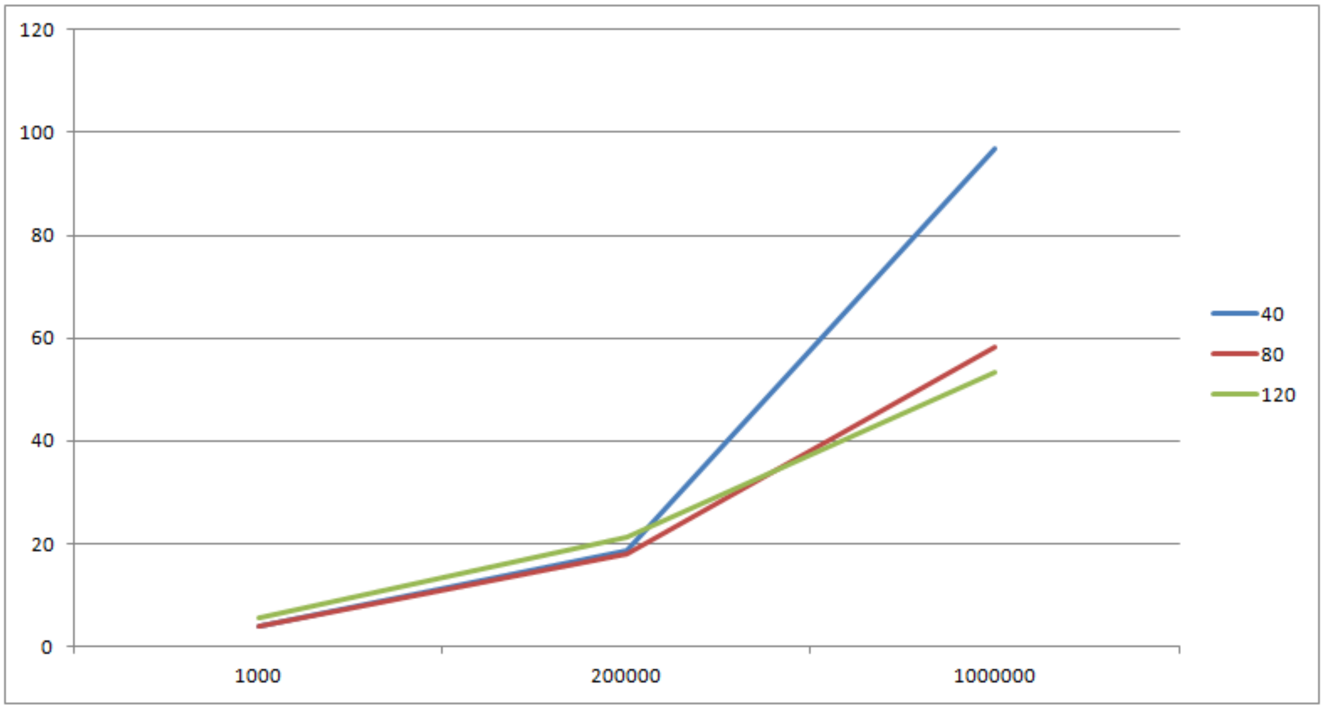

figure 1.2: Fifty Update Elapsed Execution Time (sec) vs. Population for Matrix Sizes 40, 80 and 120

Finally, we ran a number of tests with progressively increasing populations in order to try to show the gradual degradation of our system due strictly to increasing population values. From figure 1.3 we can see that at low populations, having more hosts is actually detrimental. This is due to the additional overhead of passing population values between more hosts when the extra computing power is not needed. Basically, at low populations, more hosts only serves to add network delay with no increase to execution speed. Once the population raises past 12,000 we can clearly see that the additional hosts are indeed improving the execution speed of the system and that these gains clearly increase further the larger the population gets. It is interesting to note the strange results at the 30,000 population level where the 4 host case is apparently running better than the 9 host case. This event clearly appears to be an anomaly when comparing with the rest of the gathered data and can likely be attributed to the environment in which we performed the tests (i.e. the CSIL computer lab). One final point that is interesting to note is that the tests for a population of 1000 are slower than those for population 2000. One reason we have suggested for this behavior is that since it was the first run in each case, the initial startup time was increased for these.

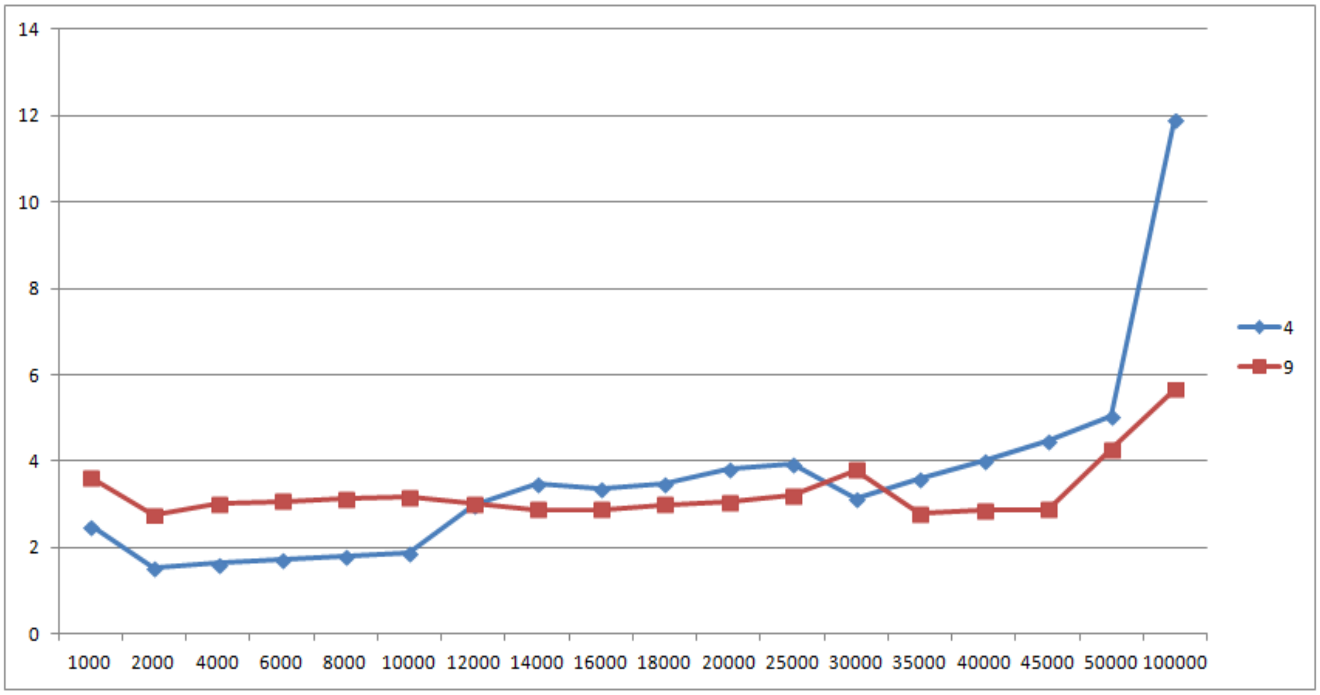

figure 1.3: Fifty Update Elapsed Execution Time (sec) vs. Population for 4 and 9 hosts

Distributed System Issues, Problems, and Pitfalls

Our first issue was with multithreading. Strange memory errors and segmentation faults in development of initial multithreaded program were arising unexpectedly. We solved this by adding more locks to various critical sections to ensure that the memory being accessed is not corrupted by being adjusted at the same time by two different threads.

Our second issue was with multi-hosting, thanks to C sockets. For an extended period of time, we did not know that all the data on the socket would not be read into the buffer in one go, so it was sporadically throwing off our inter-client communication. After reading up on C sockets, we discovered that the application is responsible for ensuring that all data is read from a socket, rather than the ‘recv’ function itself, so we found out we had to loop until all of the data was actually read, not just the first chunk.

Team Work, Distribution, and Development Process

As a team, we were able to do the majority of our work in our weekly, bi-weekly, and by the last two weeks multi-weekly team meetings, and it was very fortunate that the entire team was available for these meetings. In addition to these meetings we also remained in regular contact with email as well as Facebook. By keeping up with all of our team members using these various communication methods, we were able to stay on the same page as we developed our program. Furthermore, we used a group SVN repository for version control of our program that allowed us to maintain a good record of changes made to our program. The work was fairly well distributed and in the end we seemed to fall into these general categories:

Mike Lim: Project Leader, Universe Networking, Protocol Design, Main Server, Visualization, Dungeon Master.
Karl Olson: Visualization, Algorithm Analysis.
Tim McKenney: Main Server, Visualization.
Jared Daley: Universe Networking, Protocol Design, Main Server.
Cody Hart: Universe Networking, Protocol Design.

Our group has agreed that, if the group was given 100 total points to divide up amongst members for work on the project, Mike would receive 24 points and the rest of us would receive 19.

If We Could Go Back a Month, How Would We Approach It?

One thing that still resonates with our team is the fact that we were told not to plan everything and instead just start getting to work. While we took this to heart on the first day of our group being together by knocking out the multithreading of the universe before most of our classmates even thought about joining a group, project planning became an integral part of our solution. For the next few weeks, every time we met, we would whiteboard the solution to our next problem and spend a lot of time discussing how we would go about implementing it. While this helped clear up any discrepancies between group members and the task at hand, and no doubt led to a better overall system, in retrospect it cost us a lot of time that could have been used to implement features we didn't end up getting to (most importantly, load balancing). However, we take this with a grain of salt because had we spent less time planning and whiteboarding our solutions and more time getting down to business, there is no guarantee that it would have been a better route to take. It is easy enough to say we would have gotten more done had we spent less time planning, but until we actually go back a month and redo it, it is impossible to say which route would have been optimal.

Future Potential Development Plan

I. Load Balancing
As mentioned in the section above, load balancing was definitely one of our intentions as far as features we were unable to complete. One of our goals was to get the system up and running on around 36 hosts or more. The problem we had with this was that operating a secure shell on the majority of the Linux lab was unreliable, especially at peak hours of the day. The odds are with a very large number of hosts one of them will bog down the system (or be restarted mid-simulation) which defeats the purpose of using so many hosts; and without load balancing, our system can't recover itself from a bottleneck host or a host that drops out. This limited us to using around sixteen instances of the universe and potentially hurt our maximum overall speed.

II. Improve Cleverness of Robot Data Structure
Another problem we intended on tackling was the fact that even though our system was making use of clever data structures to turn an $O(n^2)$ set of operations into a smaller sum of $O(m^2)$ calculations, we could have improved on that by turning the $O(m^2)$ into an even smaller $O(k^2)$. To do this, we needed to improve on the cleverness of our data structure implementation (and possibly remodel the whole thing) by having the first robot in a cell do its distance calculations and then pass that information to all the other robots in the cell, so that subsequent robots in the cell would not have to recalculate their distance with that robot. The next robot to do its calculations would also pass that information to all the other robots in the cell and so on. As you can see, by the time you get to the last robot, it doesn't need to do any calculations because all the other robots have given it the information it needs. This would reduce the number of intra-cell calculations from m^2 to $(m^2 / 2)$. Unfortunately, we did not have enough time to implement this.

III. Smarter Distribution of Threads-To-Cells
On day one we naively implemented our threading so that each grid cell in the universe would have its own thread. As you can tell, without multi-hosting, this can lead to a very large number of threads generated for the universe. We improved on this by dividing by the number of universe clients in a multi-hosted system and then distributing those threads amongst each host. For example, originally a grid size of 64x64 would have generated 4096 threads. With a set of sixteen hosts, this number is reduced to 256 threads per host -- this is still too many. We intended on

improving this further, but didn't have the time to get around to it, so we may have lost some performance due to thread scheduling.

IV. Improve Bounding Box

Firstly, a bounding box is what we used to detect if a robot is within reach of a given cell. A virtual square around a robot's field-of-view is our implementation of the bounding box. If this square crossed over the boundaries of a grid cell then it is considered to be within reach of that cell and the robot has to perform its distance calculations with all the robots in that cell. To improve on this we wanted to turn our square bounding box into a "field-of-view" bounding box - a bounding box which surrounds a robot's field-of-view - so we would know absolutely if a robot is within reach of a given cell, and therefore could be interacting with its robots, rather than just assuming it is based on a generous square surrounding the entire robot.

V. Spin the System Up On Amazon EC2 Machines

Right out of the gate our group was talking *ec2* machines for the multi-hosted environment. Little did we know it was going to be a lot more difficult to implement the networking portion and have it functioning correctly on a set of CSIL machines, let alone some machines on Amazon's server which none of us had great experience with. With about a week to go, our system was functioning pretty well on the CSIL machines and we tossed around the idea of trying them on *ec2*, but we figured our time was going to be better spent by polishing up some pending issues with our current local system rather than invest valuable time trying to spin the system up on Amazon.

VI. Improve how StartupServer Handles Dropped/Excess Clients

Currently, if one of the startupClients connected to the startupServer are dropped, the server and all the clients must be stopped and then the server started up again and all the clients must then rejoin. Clearly this is not an ideal situation and in the future we would like to allow the server to stay up and running should one of the clients drop out and allow a new client to join the server. Furthermore, we would like to allow more than the 'n' required clients to join the server initially and be put in an 'inactive' pool. Then, should one of the clients drop out, we can immediately assign a new client from the 'inactive' pool to the server. This would not only save the need to monitor the startup server/client structure as closely, but would also allow for much more robust error handling due to network or other hardware issues.

VII. Enhance Graphics Functionality and Efficiency

There are definitely some issues with the graphics solution as implemented. The sprite rendering, while faster than vector rendering, changes the angle of the sprite based on the top of the sprite rather than the center of it. That behavior is incorrect. In a later version, it would be ideal to move solely to a position corrected sprite system, as it would combine accuracy with lower system overhead. It would also be worth considering dynamic compression/decompression of robot locations as the population size begins to climb. This would hopefully allow for more if not all cells to be actively displaying robots. While the histogram scales fairly decently with population and matrix size, it would be useful to have keyboard commands that meaningfully allowed for the average population used to determine the color range in the histogram to be modified or scaled in real time to help make the visual data more meaningful. It might also be useful to incorporate world size into those histogram calculation options as a way of showing sparseness.