An Abstract Parallel LGA Flow Simulation

Charlie Thornton

CIS 690: Implementation Project
Kansas State University
2002

TABLE OF CONTENTS

TABLE OF FIGURES ..... **iii**
INTRODUCTION **1**
PROBLEM DEFINITION **1**
SOLUTIONS **3**
FIRST SOLUTION 3
FINAL SOLUTION 8
TESTING **11**
RESULTS **12**
DISCUSSION **16**
WORKS CITED **17**

TABLE OF FIGURES

Figure 1: The Von-Karman Alley Problem 2
Figure 2: Slip and No-Slip Boundary Conditions 2
Figure 3: Program Architecture 4
Figure 4: Code Listing for GridBlock Run Method (Version 1) 4
Figure 5: Cell Structure Overview (Version 1) 5
Figure 6: Cell Structure Details (Version 1) 6
Figure 7: Tranfer Method Code 7
Figure 8: Code Listing for GridBlock Run Method (Version 2) 8
Figure 9: GUI Time Step Control 9
Figure 10: Cell Structure Details (Version 2) 10
Figure 11: Early Testing with Dots 11
Figure 12: Vector Field Representation of the Grid 12
Figure 13: Timing Analysis on Sunflower 100x100 Cells 13
Figure 14: Timing Analysis on Sunflower 500x500 Cells 13
Figure 15: Results 200x300 TS: 0 14
Figure 16: Results 200x300 TS: 50 14
Figure 17: Results 200x300 TS: 100 14
Figure 18: Results 200x300 TS: 200 14
Figure 19: Results 200x300 TS: 400 14
Figure 20: Results 200x300 TS: 1000 14
Figure 21: Results 200x300 16 Threads TS: 400 15
Figure 22: Results 300x200 Multiple Boundaries 16

INTRODUCTION

The field of cellular automata is a rich hunting ground for intuitively parallel programming problems. Each cell in an automata can be considered a black box to every other cell; making decisions by its own rule set and caring only about the output of neighboring cells. One interesting application of this natural parallelism is lattice gas automata (LGA). LGA simulations were developed to efficiently approximate fluid motion as described by the Navier-Stokes equations [8]. Lattice gas models work with the notion of so-called macroscopic fluid particles [8]. The behavior of the underlying system can be determined by modeling the behavior of these high-level particles.

A lattice gas model consists of a lattice of sites connected by channels. Virtual fluid particles travel between sites via these channels. They can be visualized as gas particles flying unobstructed until colliding either with each other or a contrived obstacle [1, 2, 4, 5]. The resulting collisions and the shape of the lattice are the focus of LGA models. Many different rule systems have been developed to determine the outcome of particle collisions. The original rule set, designed for a square lattice in 1973, is named the HPP model after its creators: Hardy, Pomeau and de Pazzis [8]. Since then, numerous lattice formations have been explored [4], each requiring a different set of rules.

Cellular automata are intuitively parallel, and LGA are no exception. Each lattice site requires only the output of neighboring sites to calculate its next state. There is no need for a barrier, allowing for very effective performance improvements as more processors are added to the computation. Their natural parallelism makes it possible to compute very complex fluid simulations relatively quickly.

A strong disadvantage of this project is that it is an abstract representation of reality. I was unable to locate information on linking the output of any form of lattice gas automata with real values. This presents a heavy burden of uncertainty. Certain factors such as particle density correspond to different fluids, but I could not find any further information on that relationship. Visually, the output of the simulations described in this paper match reality.

PROBLEM DEFINITION

The program described in this paper visually approximates a solution to the Von Karman Alley problem described by Hannon [3]. The Von Karman Alley is a laminar flow around an obstacle between parallel plates (Figure 1).

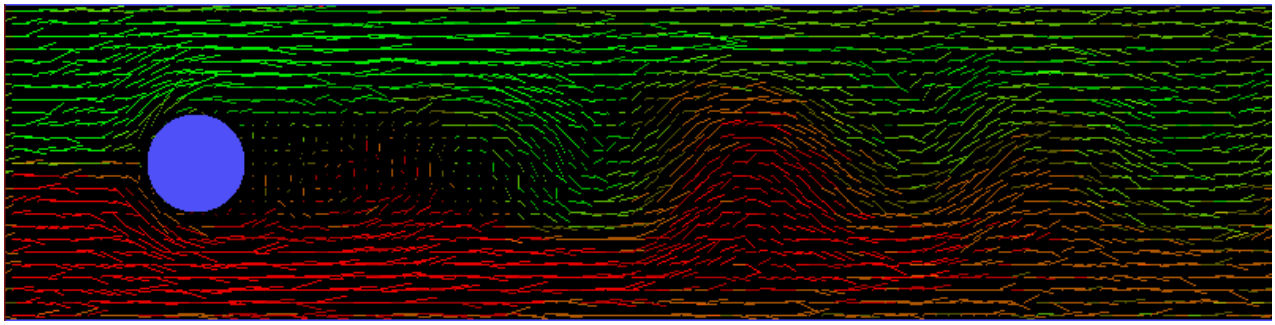

Figure 1: The Von-Karman Alley Problem

Taken From: *Lattice Gas Automata: Von Karmon Alley*. Retrieved on 6/28/2002. David Hannon. Available at: http://poseidon.ulb.ac.be/simulations/vonkarman_en.html

The left wall of the alley creates a flow directed to the right, parallel to the top and bottom plates. The right plate sinks the flow (by removing fluid particles from the model) as it arrives. The top and bottom walls of the model are considered slip boundary conditions [3]. The obstacle is a no-slip boundary. Figure 2 illustrates three different particle collisions using slip and no-slip boundary conditions.

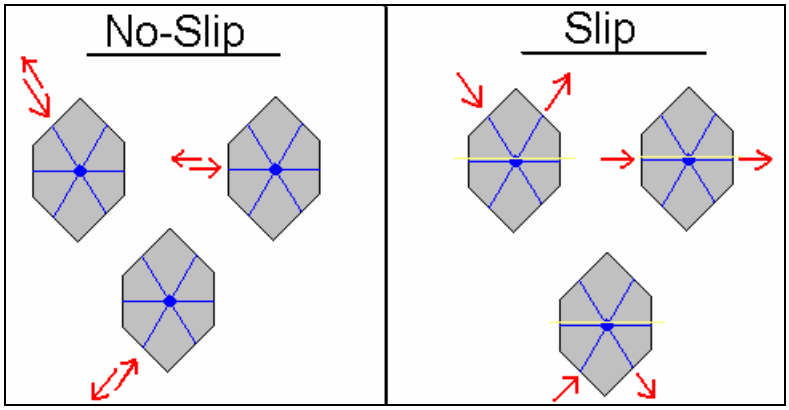

Figure 2: Slip and No-Slip Boundary Conditions

The no-slip condition always repels incoming particles along the channel from which they entered [1]. The slip condition mimics reflection off a surface. Slip cells must be initialized with a specific reflection axis. In Figure 2, particles bounced off the cells' horizontal axes, but slip cells could be configured to reflect particles off the vertical axis.

The collision rules for intermediate (ie. not boundary) cells follow the FHP-III rule set [1]. The FHP (Frisch, Hasslacher, and Pomeau) model was first presented in 1986. Two important advantages of the newer model are a hexagonal lattice and the presence of resting particles. Buick describes each model variant in detail as well as the FHP-III model's mathematical relationship to Navier-Stokes [1]. The FHP-III variant is the most robust of the three, and it places the fewest limitations on particle collisions.

Conveniently, there are 128 collision states, allowing an implementation that represents the state of each cell in the model with a single byte. A table of all in- and corresponding out-states is given in [1].

The challenging element of this project was to run clumps of cells in parallel. From this point on, the collection of all cells in the model will be referred to as the grid, and the parallel sections of the grid will be referred to as grid blocks. Ideally, the user should be able to choose both the number of cells in the grid and the number of grid block divisions. Also, the user is responsible for advancing the simulation an arbitrary number of time steps. Allowing the user to advance the simulation manually allows for a more visual simulation as well as eliminating convergence tests within the software.

SOLUTIONS

The following sections discuss two slightly different solutions to the Von-Karman Alley problem and provide a comparison of the two. Most of the software will be described in the first solution, as the second solution is only a variant of the first. The second solution was optimized for multi-threaded performance.

FIRST SOLUTION

The software I created to run the LGA simulation for the Von-Karman Alley consists of a graphical user interface (GUI) and an underlying data model. I chose a document-view architecture to coordinate the behavior of these two system components. Figure 3 shows the relationship between the GUI and underlying data model.

Figure 3 shows the program's communication architecture. The Run class is the entry point for the program and simply populates Document with the necessary classes before relinquishing control. The classes on the right are the main program frame and dialogs. They are used by the document to modify the underlying data model. Note that all communication takes place via the document. Also, in Figure 3, creates implies uses. The classes on the left encapsulate the grid that will be used for the simulation. Document calls methods within Grid, Grid passes commands down to GridBlock, and the grid blocks work directly with the cells. The document-view architecture is not perfect in that Grid draws itself. For simplicity, Grid extends JComponent and each grid element, including all cells, contributes something to the drawing process. Grid is simply inserted into the center of FlowFrame's content pane, allowing repaint calls to refresh the image automatically. An unfortunate drawback of this simplicity in the current implementation is that graphics calls interfere with the execution of time steps in the threads. Forcing the main program thread's call to *updateViews* to wait until all worker threads have finished their jobs could repair this flaw.

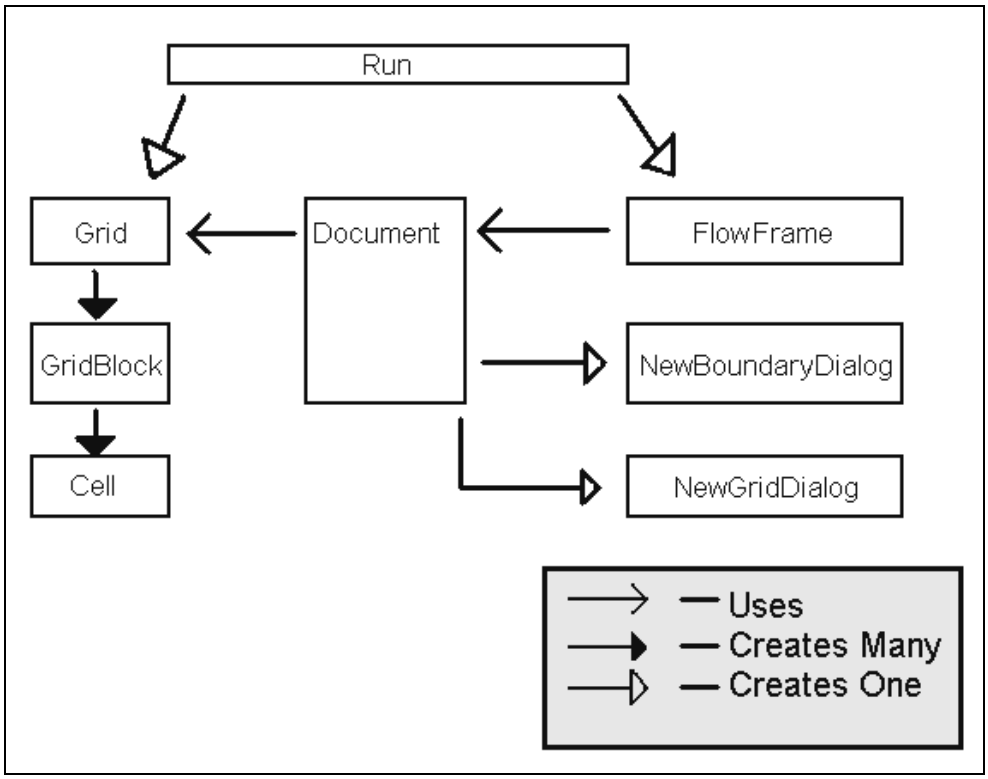

Figure 3: Program Architecture

The parallel elements in this program are the grid blocks. Class GridBlock extends the Java class Thread. Grid blocks consist of a rectangular section of cells in the grid. Each cell is connected via channels to corresponding neighboring cells. There are as many channel variables created along the borders of blocks as there are actual channels in the lattice along that section. At each time step, the grid blocks are allowed to pass a semaphore and update all the cells in the block. Figure 4 shows the method in GridBlock that controls thread execution. The step method called from *run* simply updates all of the

```
public void run()
   {
      while (true)
      {
         d_sema.P();
         step();
      }
   }
```

Figure 4: Code Listing for GridBlock Run Method (Version 1)

the cells in the block. The variable "d_sema" is a counting semaphore passed into the grid block at construction time and is shared between all of the grid blocks. The grid increments the semaphore n times, where n is the number of grid blocks in the grid, for each time step.

Each grid block is composed of many cells. The autonomous behavior that drives the simulation takes place at the cell level. Figure 5 shows the structure of the cell objects.

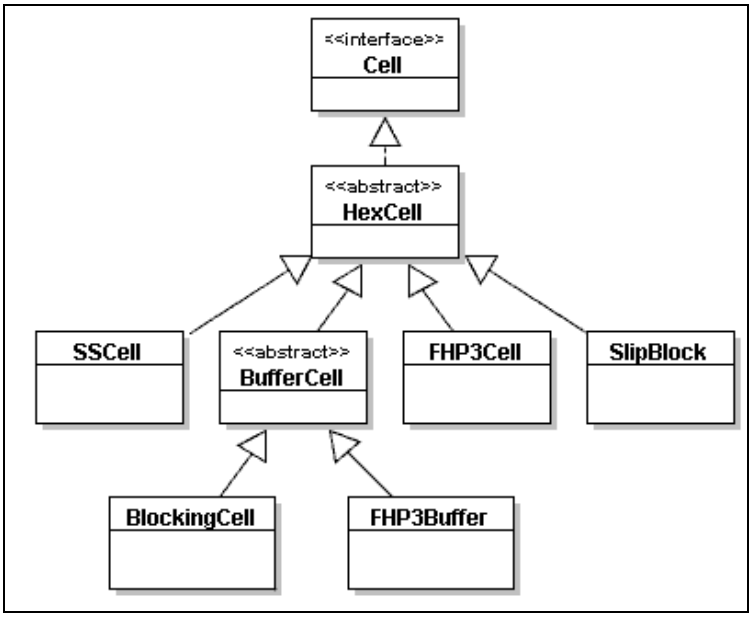

Figure 5: Cell Structure Overview (Version 1)

In order to simplify the program, the behavior of the cells is controlled through inheritance. This eliminates the need to keep track of which cell is where. The Cell interface defines the methods shared by all cells. The grid blocks simply maintains an array of type Cell, and any time an obstacle or a boundary condition needs to be inserted or removed from the model, the subclass implementing the Cell interface is inserted into the array.

The HexCell class shown in Figure 5 is an abstract class extended by all of the cells in the simulation. It implements most of the methods necessary for a hexagonal cell with six particle paths to neighboring cells. Figure 6 shows a detailed description of the methods implemented by each subclass of HexCell. A careful examination of Figure 6 would reveal that the subclasses are only responsible for defining the *collide* and *draw* methods guaranteed by the Cell interface. The *collide* function describes the particle behavior within a given cell. Within the FHP3Cell, for instance, this function examines the current state of the cell and then produces the next state using the FHP3 collision table located in Buick [1]. FHP3Cell defines a static lookup table to implement the collision function. SlipBlock's *collide* function examines the current state and generates a new state with particle velocities reversed. As a convenience, each cell also implements a method (called draw) to draw that cell to a graphics device. Again, this provides a simple way to control how individual cells are displayed, without the grid blocks knowing anything about the underlying instances of Cell objects.

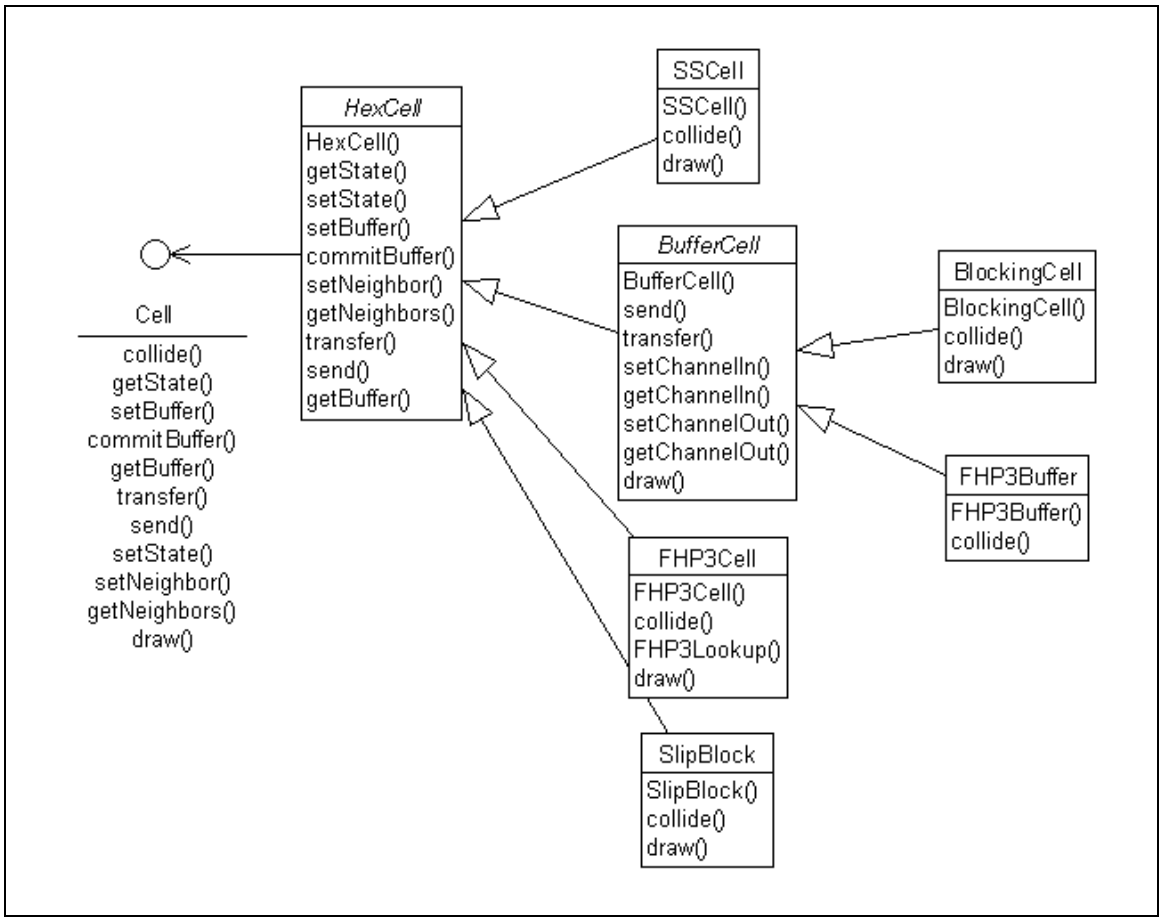

Figure 6: Cell Structure Details (Version 1)

Figure 6 also clarifies the purpose of the abstract class, BufferCell. Buffer cells are used on the edges of grid blocks to communicate between threads. The extra getters and setters for channels are used to weave together adjoining grid blocks with synchronized boolean channels (called ChannelBoolean in the synchronization package). Each pathway on every buffering cell that binds two BufferCells together is implemented as a boolean channel. Thus there are many channel objects in a large multi-threaded model.

It is assumed that all particles (except any resting particles) are traveling inward when *collide* is called, and following the call they are traveling outward. The transfer method shifts outgoing particles to neighboring cells. An ugly necessity in the transferring process is that transfer must not destroy the current state of neighboring cells. This allows those cells to perform their own transfers accurately. After every cell has sent out its contents, *commitBuffer* is called to finalize the transfer and commit the changes to the cells' states.

The state of a cell is represented as a single byte to save memory. Six bits are needed to represent particles traveling on the paths. A seventh bit is nessesary to represent a resting particle. This method of data packing also makes it simple to handle the transferring of values between cells using bit-wise arithmetic. The performance enhancements that

would normally have accompanied this method are questionable under Java, but this technique, while at first cryptic, simplifies the code considerably. Figure 7 is an example of the transfer code found in HexCell.

```
byte mask = 0;
    for (int i=0; i<6; ++i)
    {
      mask = (byte)(1 << i);
      if (neighbors[(i+3)%6] != null &&
         (neighbors[(i+3)%6].getState() & mask) != 0)
      {
        d_buffer |= mask;
      }
    }
```

Figure 7: Transfer Method Code

Figure 7 shows the section of code that deals with incoming cells at transfer time. Each cell has six neighbors. The first six bits in a cell's state represent unit velocity vectors of particles within a cell. If the cell located down and to the left of the current cell contains a particle traveling up and to the right, the result of transfer should be to assert the up and to the right velocity bit of the current cell. The code in Figure 7 contains a bit mask. The mask can be though of as the current direction vector. In the first iteration of the loop (i == 0), the up-right bit of mask is set to 1. Then the lower left neighbor's state is compared against the mask. If the lower-left neighbor contains a particle traveling up and to the right, the current cell's buffer is ORed with the mask. On the next iteration the mask is set to represent a particle moving to the right and the cell to the left of the current cell is checked.

There are several flaws in this solution. The most detrimental flaw is the method by which program flow is coordinated in the threads. The counting semaphore shown in Figure 4 is incremented once per thread each time step. In addition to creating a preponderance of synchronized method calls, this technique created a barrier effect among the threads. No grid block could advance more than a single time step beyond any other block. Another performance-related disadvantage was the way in which the grid blocks were linked together. Each cell on the edge of a grid block contained at least one shared communication channel. This added up to an enormous amount of message passing between the grid blocks to execute a single time step. In addition to being a performance issue, this architecture would be very difficult to translate into a distributed system. These issues are addressed in the final solution.

FINAL SOLUTION

The final solution to the lattice gas simulation addressed the major performance and scalability issues detected in the first version. The most dramatic enhancement was to ease the restriction on grid block time step advancement. In this version, a block may advance one step beyond its neighbor's current time step, two steps beyond the blocks that touch the neighbors, and so on. In general, blocks advance when the channels connecting them to other blocks are populated with the necessary neighboring state information. At the beginning of a time step all blocks put their state information over their outgoing channels. Then they get information from their incoming channels. Once all neighbor information is retrieved they calculate their next states and repeat the process. The blocks run in this fashion for a set number of time steps before waiting for another instruction to begin. Figure 8 shows the final version of the run method found in GridBlock.

```
public void run()
  {
    int n;
    while (true)
    {
      n = d_stepChannel.look();
      for (int i=0; i<n; ++i) step();
    }
  }
```

Figure 8: Code Listing for GridBlock Run Method (Version 2)

The code listed in Figure 8 shows a mysterious *look* operation being performed on an integer channel. This operation allows each grid block to observe the value stored in the channel without forcing it to empty. Threads must register as watchers within the channel in order to observe the stored variable in this way. Once all registered watchers have looked at the variable, the channel is emptied and future calls to look are blocked. Progress for looking and getting is halted until a new value is put into the channel.

The instruction to begin executing time steps is strongly linked to the user interface. In the lower, right hand corner of the GUI (see Figure 9) there is a button and text box that control time step execution. When the button is clicked, a channel (in Grid, referenced by all grid blocks) is filled with the integer value contained in the field (1 when the text box is empty). This channel has been modified to allow an extra operation. Using a look operation (see Figure 8), the grid blocks can determine how many time steps to execute. These improvements eliminate the barrier effect on the simulation.

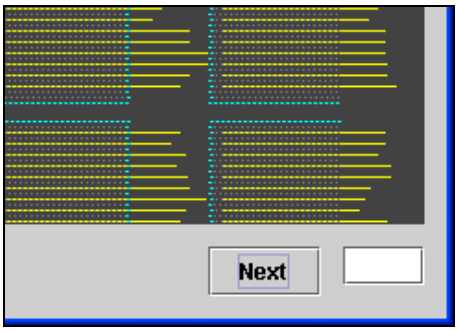

Figure 9: GUI Time Step Control

An underlying point in the previous description was the modification to the way particle traffic information was shared between grid blocks. In this solution, cells no longer contain any channels connecting them to cells in other grid blocks. All of the channeled communication takes place at the grid block level. This method dramatically decreases the number of channels required to communicate between the grid blocks. For each of the eight (four corners plus four sides) directions a grid block needs to communicate there is a channel. Channels allow grid blocks to transmit arrays of integers representing the states of cells on the borders. In addition to the performance improvement offered by the reduction in synchronized calls, this change creates an architecture that is readily scalable to distributed computing.

The communication change necessitated an overhaul of the cells' architecture. Cells are no longer responsible for communicating over channels with neighboring cells. Instead, the buffer cells transfer their states into the grid blocks for holding until the data is transferred, and OR the data stored in grid block for neighboring cells with their current states. This method employs the method found in Hansen [6] to force the buffer cells to overlap with one another on the thread boundaries. Essentially, the two columns of cells on either side of any thread border form a single actual column of cells in the simulation. Figure 10 shows the modified version of the cell structure. The focus of this figure is the abstract BufferCell class. It now provides the functionality of overlapping with neighboring buffer cells and allows subclasses to plug in their own collide function.

Figure 10 also shows that the SSCell class does not have a version that extends the BufferCell abstract class. Until such a class is added, no boundary conditions can be inserted into the body of the model. The implementation of this feature was restricted only by time. I had no subtle motives for omitting it.

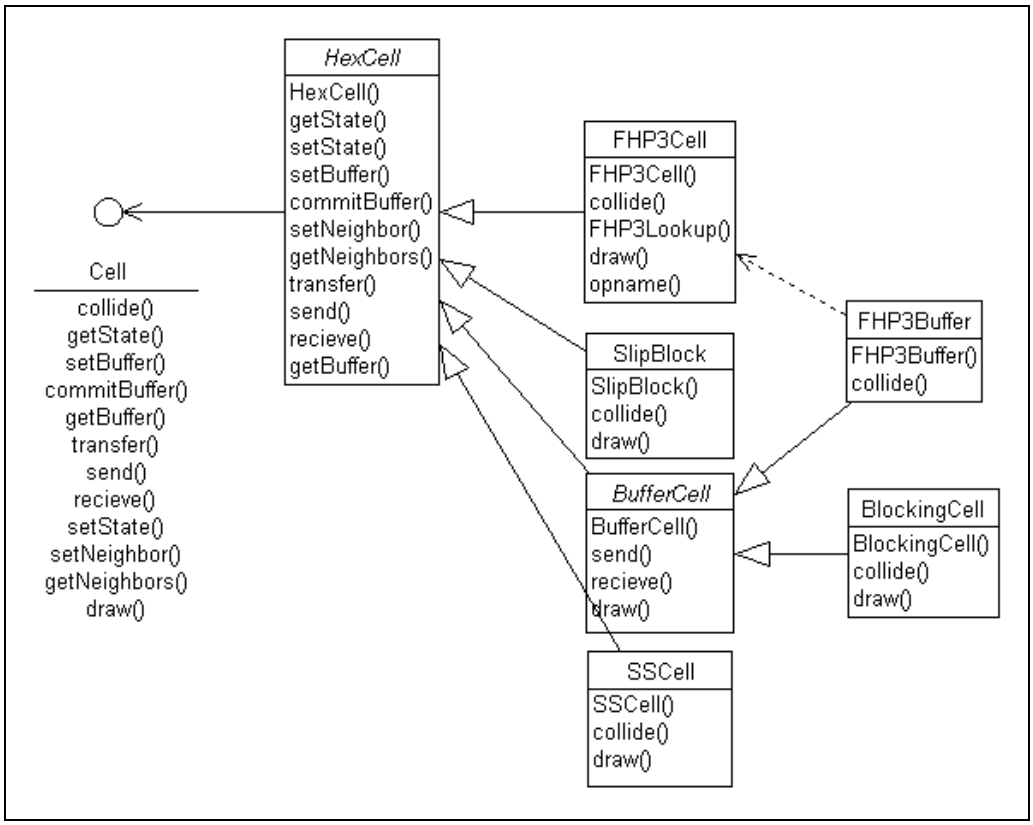

Figure 10: Cell Structure Details (Version 2)

The grid, previously responsible for binding individual cells together, was simplified somewhat by the change in grid block communication. The grid is still responsible for arranging the channels, but in the final version the algorithm is simpler. The program iterates through all but the outermost grid blocks and binds them together at all eight points (four corners, four walls). There is some tricky code at the grid block level to deal with the duplicity of a corner cell belonging to both a wall and a corner. The algorithm in grid block's *step* method (executes a single time step) consists of the following steps:

1. Copy cell state data into outgoing arrays
2. Synchronize outgoing array endpoint states with outgoing corner states
3. Put data out over all existing channels
4. Get incoming data from all existing channels
5. Synchronize incoming array endpoint states with incoming corner states
6. Move the data stored in the grid block to the individual cells
7. Cells communicate and update states (particle transfer)
8. Cells calculate next state using the collide function (particle collisions)

The heart of the algorithm is the final two states. Particles travel, and then collide. This process is repeated for each time step.

TESTING

The first tests for the program were very primitive. I created very small grids of data and dumped matrices of the state values of each cell in the grid to a file for several consecutive time steps. By comparing the states of various cells between time steps, I was able to ensure that particles were obeying the FHP-III rules in the cases I could check. This process induced sufficient tedium to motivate me to develop a graphical user interface early in the program's development. At first, the GUI drew a dot representing each occupied cell in the model. Figure 11 shows an example of this drawing method.

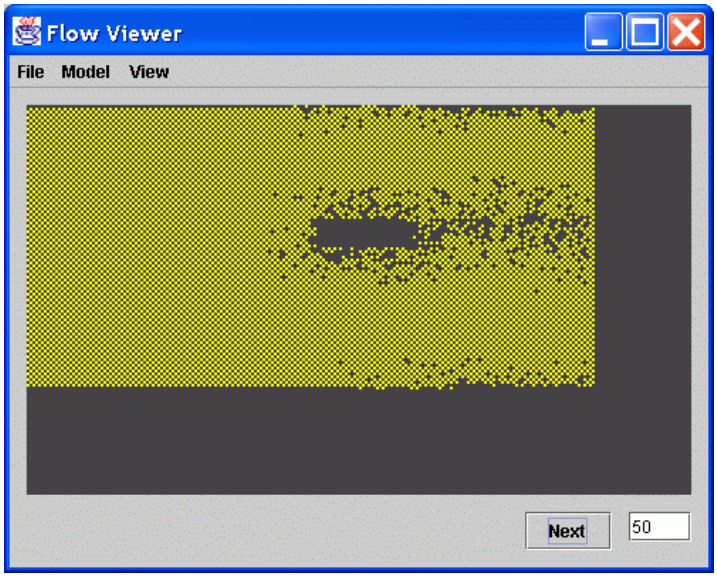

Figure 11: Early Testing with Dots

While it was helpful on sparsely populated models the dot drawing method was obviously not useful when the model became dense enough to observe many collisions. The grid is best represented as a vector field. The final version of the GUI (Figure 12) shows this representation. The yellow lines seen in Figure 12 are the net velocities of particles within 5x5 neighborhoods of cells. The color of the vectors is always the same. Their direction and length change to represent the neighborhood. The fields calculated by this program are somewhat sloppy in that they allow overlap between cell neighborhoods. This overlap helps to smooth the appearance of the results.

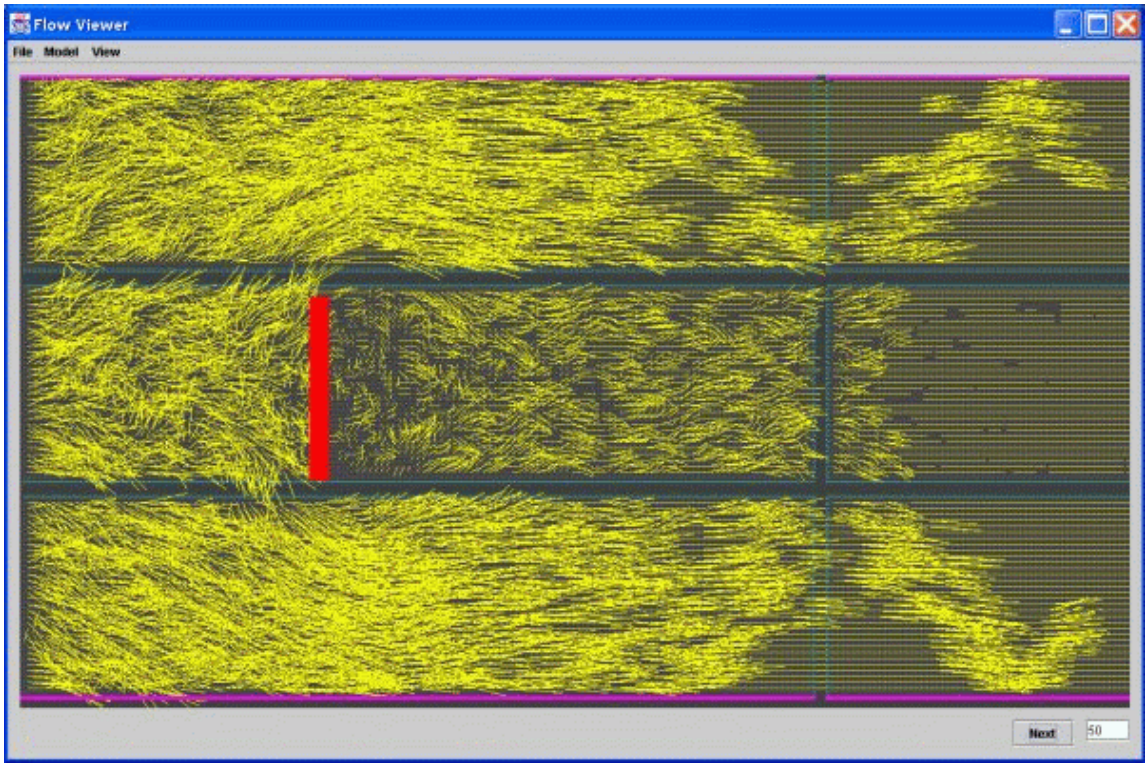

Figure 12: Vector Field Representation of the Grid

RESULTS

This section is a collection of results from the timing of the program as well as images captured from various states of the simulation in action.

A timing analysis performed on one of the multiprocessor machines (sunflower) found in the Kansas State University's Computing and Information Sciences department's network produced the results found in Figures 12 and 13. Sunflower is a 4-processor machine. There are two interesting phenomena revealed by the figures. The first is that the difference in performance between using a single thread of execution and using four is almost exactly a factor of four. Even in the smaller grid (100x100), the overhead of using multiple threads does not outweigh the benefits. The second interesting point is that using sixteen threads did not perform as well as using four. In all likelihood, had this simulation not been able to dominate all four processors on the test machine, the ranking of four and sixteen threads would have been reversed since the higher thread count places the minimum restrictions on the current state of any executing time step. A count of sixteen allows the upper left grid block to be separated in time from the lower right block by three time steps. This freedom allows the simulation to carry on optimally should one of the threads be delayed by other jobs in the scheduling queue.

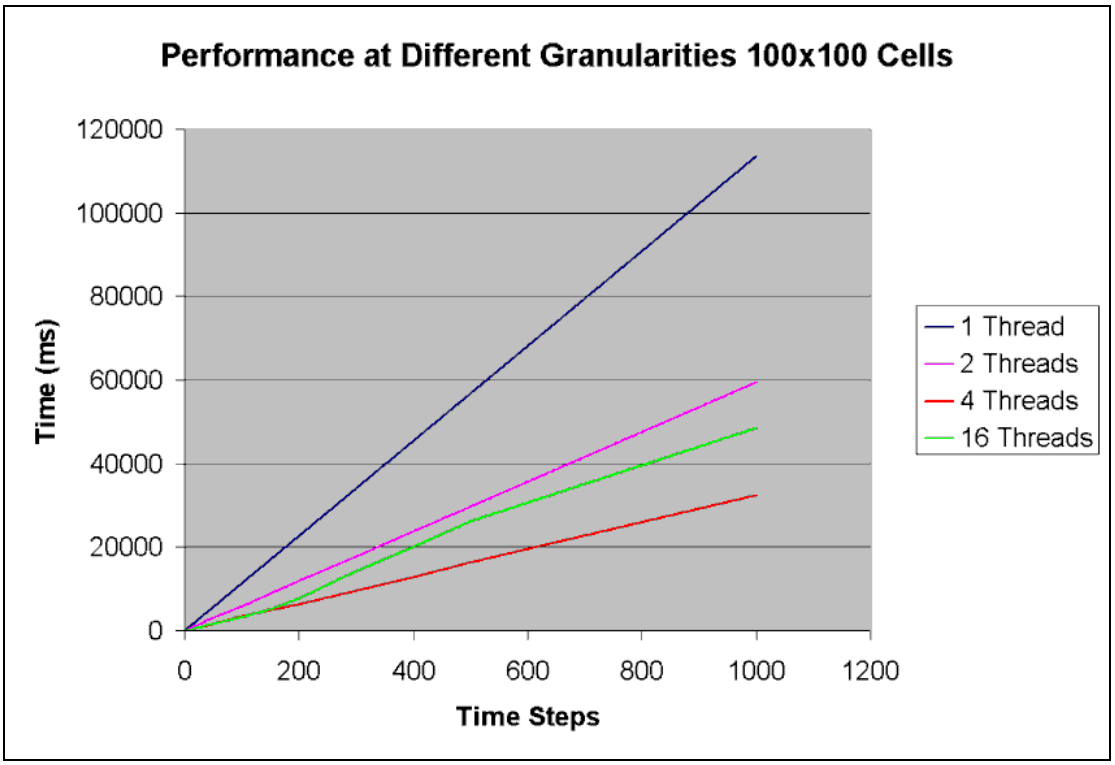

Figure 13: Timing Analysis on Sunflower 100x100 Cells

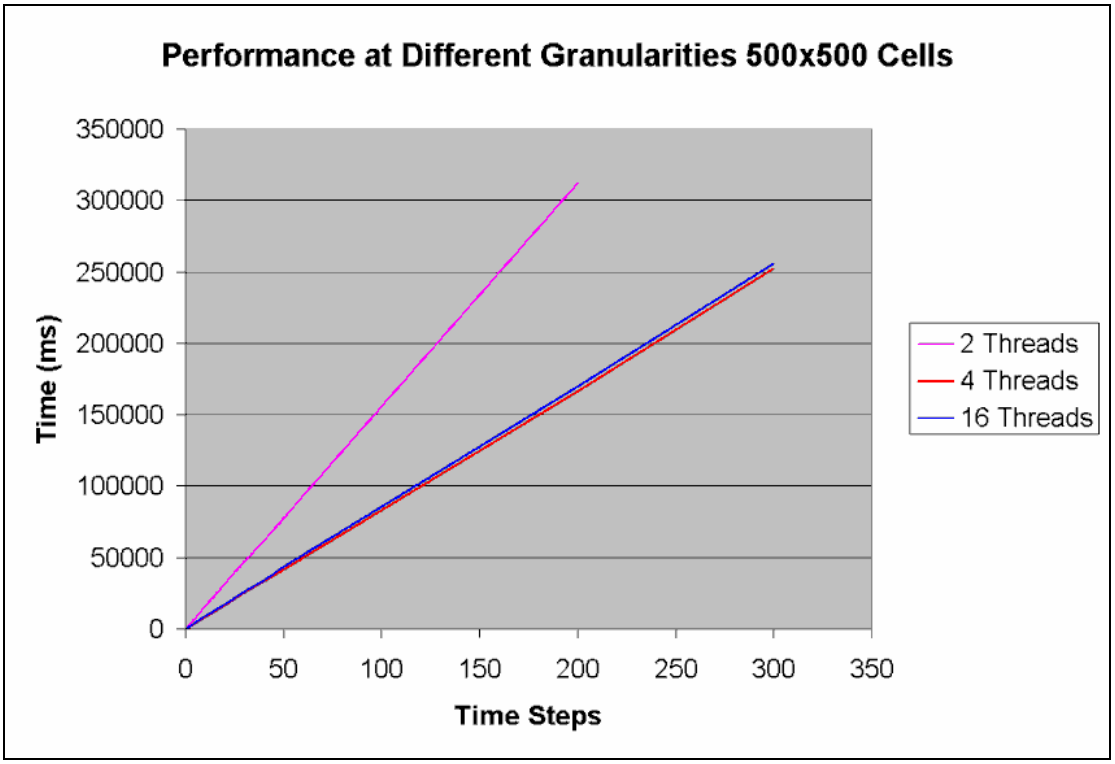

Figure 14: Timing Analysis on Sunflower 500x500 Cells

The following figures are taken from a simulation of a single obstacle within a 200x300 grid. The density of particles in the simulation is 0.28. The simulation was executed on a single thread.

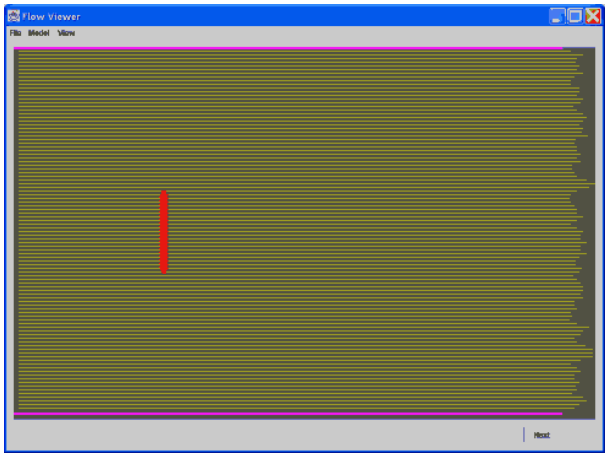

Figure 15: Results 200x300 TS: 0

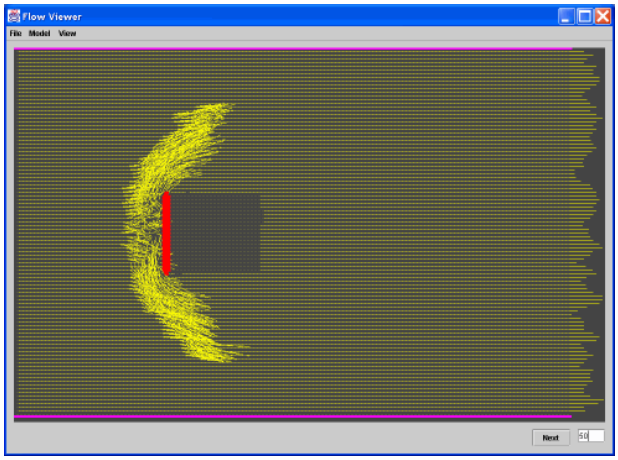

Figure 16: Results 200x300 TS: 50

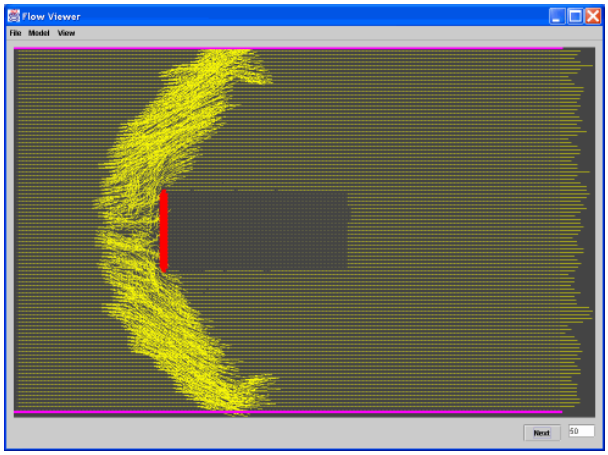

Figure 17: Results 200x300 TS: 100

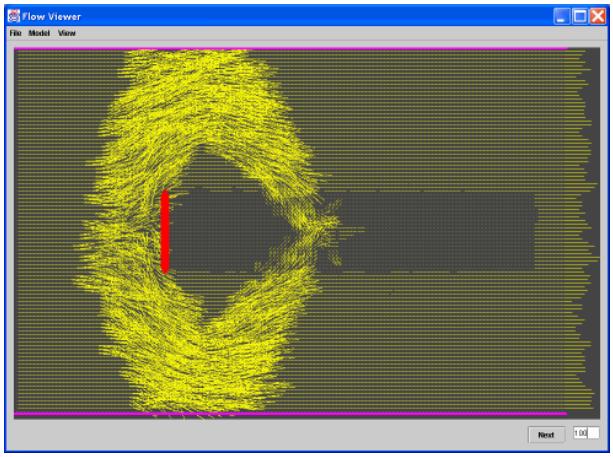

Figure 18: Results 200x300 TS: 200

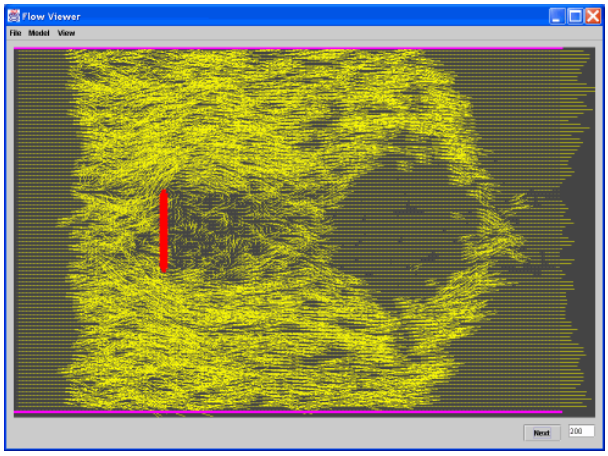

Figure 19: Results 200x300 TS: 400

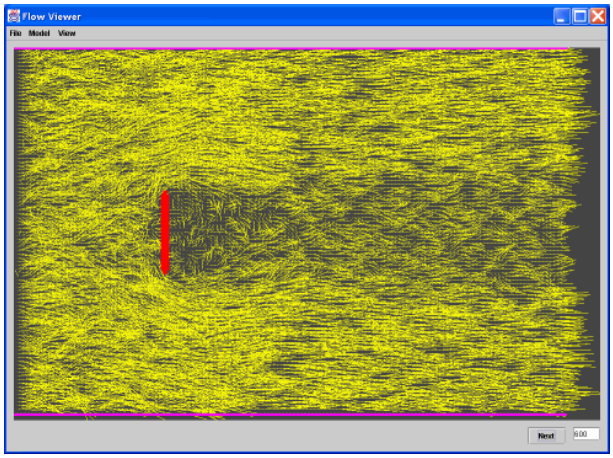

Figure 20: Results 200x300 TS: 1000

In figures 15-20 an obstacle was inserted into a flow of density 0.28, and after 1000 time steps the flows appears to have stabilized. Figure 21 shows an example of the same simulation on a grid that was divided into sixteen blocks. This image shows the state of the simulation after 400 time steps. I chose this time because it forms the most distinct image and can be easily compared to the single thread version in Figure 19. When the grid is broken up into multiple blocks the image degrades. The drawing method could be improved by avoiding a margin around the outer edges of the grid blocks. Currently, the averaging neighborhoods are truncated by the boundaries and the results appear quite a bit noisier.

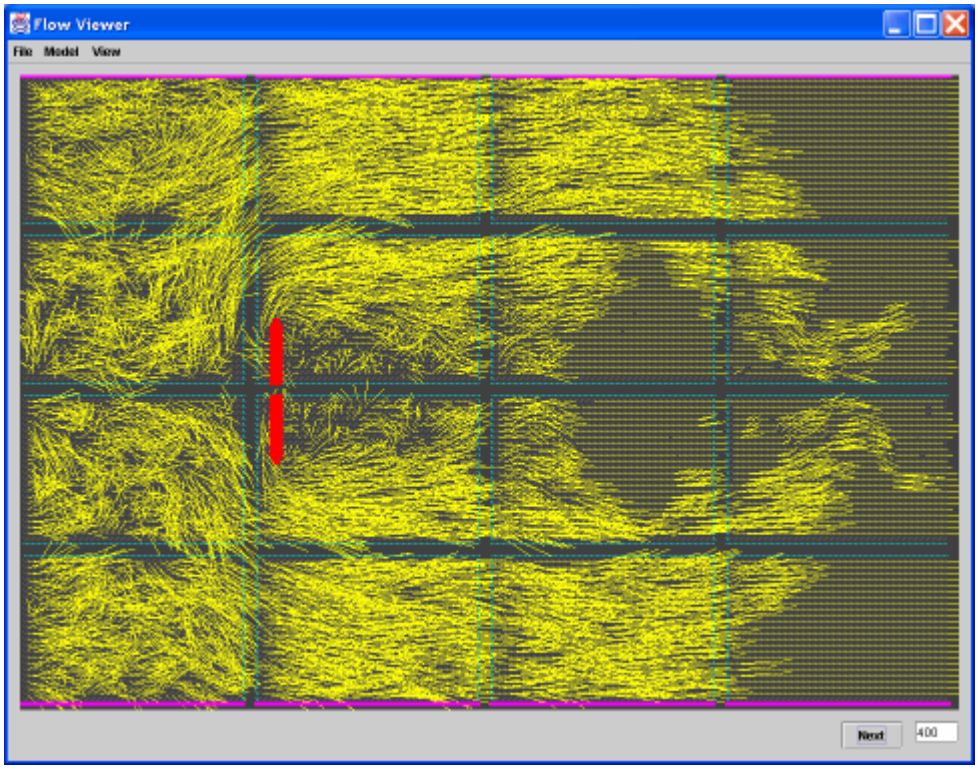

Figure 21: Results 200x300 16 Threads TS: 400

The model can also handle multiple boundaries. Figure 22 shows an example of a two-boundary model. This image was captured after 450 time steps into the simulation. Only one thread drove the simulation.

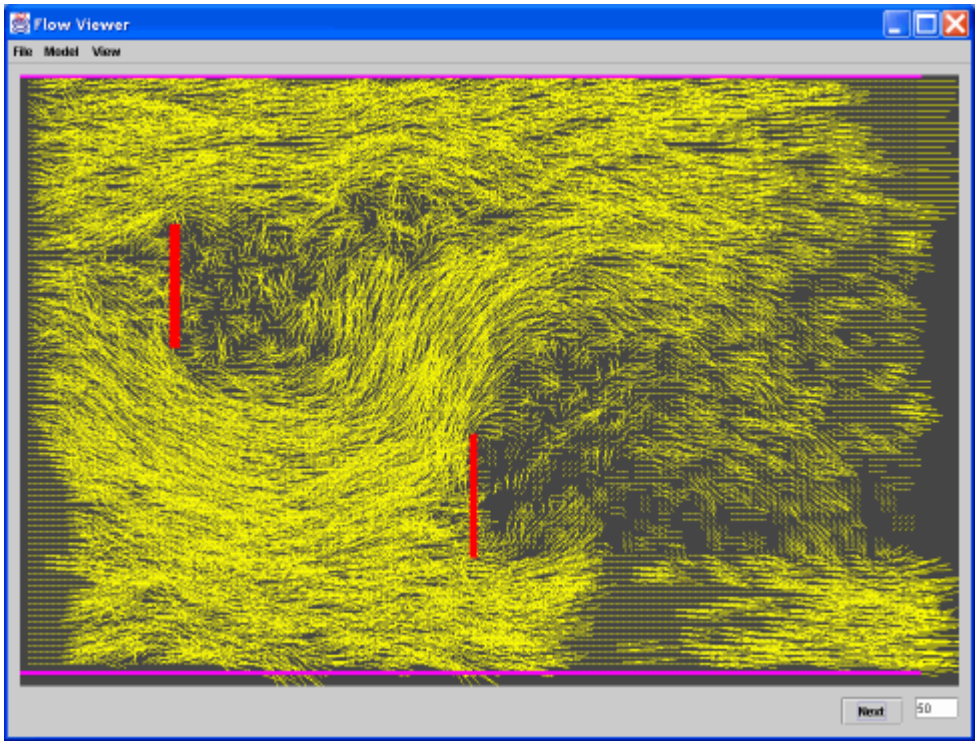

Figure 22: Results 300x200 Multiple Boundaries

DISCUSSION

Lattice gas is a noisy simulation method. Only by making larger models can we compensate for this noise. It is necessary to create parallel solutions to these problems to efficiently calculate large numbers of time steps. My results show that dramatic speedup can be achieved in parallel solutions without sacrificing accuracy. Clearly, many existing simulation codes could benefit from multithreaded implementation.

The greatest frustration of this project has been the lack of solid experimental values derived from lattice gas simulations or even instruction on modeling such values. Such a shortcoming renders this solution very close to useless from a practical standpoint. The ability to run the simulation until convergence and then find densities and velocities of the flow at selected points would make this solution far more valuable.

The next step in this implementation would be to introduce a colored tracer feature. Figure 1 showed an example of the Von-Karman alley problem in color. There is a very clear swirling to the downstream side of the obstacle. This effect is lost in my results, and I can only assume that its absence is simply an inability to properly visualize the effect. I am confident that the introduction of tracer particles would produce a similar result in this simulation.

WORKS CITED

[1] James M. Buick. Lattice Boltzmann Methods in Interfacial Wave Modeling. (1998) [On-Line] Available At: http://www.ph.ed.ac.uk/~jmb/thesis/tot.html (Last Accessed: June 2002).

[2] David Hannon and Olivier Tribel. Lattice Gas Automata. [On-Line] Available At: http://poseidon.ulb.ac.be/simulations/intro_en.html (Last Accessed: June 2002).

[3] David Hannon. Lattice Gas Automata: Von Karmon Alley. [On-Line] Available At: http://poseidon.ulb.ac.be/simulations/vonkarman_en.html (Last Accessed: June 2002).

[4] Stephan Wolfram. Cellular Automata Fluids: Basic Theory. (1986) [On-Line] Available At: http://www.stephenwolfram.com/publications/articles/physics/86-fluids/ (Last Accessed: June 2002).

[5] Franz J. Vesely. Lattice Cellular Automata. (2001) [On-Line] Available At: http://www.ap.univie.ac.at/users/ves/cp0102/dx/node126.html (Last Accessed: June 2002).

[6] P. Brinch Hansen. Parallel Cellular Automata: A Model Program for Computational Science (1993). Concurrency -- Practice and Experience 5. John Wiley & Sons Ltd.

[7] Masaaki Mizuno. A Structured Approach for Developing Concurrent Programs in Java. (1999) [On-Line] Available At: http://www.cis.ksu.edu/~masaaki/ftp/ipl99.ps (Last Accessed: June 2002).

[8] U. Frisch, B. Hasslacher, Y. Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. (April 7, 1986) Physical Review Letters. Vol. 56, Number 14