

## Parallel Simulation of Mobile Ad-Hoc Networks

### Abstract

This project was primarily a parallel computing effort and secondarily a networking project. This paper describes the challenges and solutions we encountered in creating a parallel MANET simulator. Details of the architecture are supplied, as are a discussion of other ongoing efforts gleaned from the literature.

### Introduction

*We have organized this paper to flow from (a) what we are simulating, to (b) what is currently used to simulate it, to (c) why we can't use these simulators, to (d) how our new simulator will work. Thus, discussion of MANETs and routing can be found in the first section (Motivation). The next section (Literature Survey) will confine itself to discussing simulators (e.g. PDES) in general. In the following section (Justification), we then evaluate the main PDES and non-PDES network simulators, including those not meant to simulate MANETs, and explain why we chose not to use them. These simulators were reviewed as part of our decision process on whether to write our own software from scratch, or attempt to extend existing simulators. Finally, we discuss our new simulator for the remainder of the paper.*

### Motivation / Problem Description

A network of mobile wireless nodes has significant architectural challenges, because the traditional routing and tiered control algorithms fail as nodes move in and out of range with each other [20]. A peer-peer version of this network presents even more difficulties.

It is necessary to use simulators to test and experiment with various protocols for network discovery, routing, and the content-based distribution of various messages. There exist three main simulation environments for *mobile* ad hoc networks (MANETs), e.g. GloMoSim [21], ns-2 [13], and OPNET [15]. Of these, only ns-2 has been parallelized, under the PDns project at Georgia Tech [8]. Simulations typically require excessive computational resources; as a result, software that runs on a single serial machine usually only simulates around 10 nodes [7]. Unfortunately, technology has improved to where modern networks can run into the thousands of nodes. Examples of this can be seen in the growing use of scattered sensor networks, and in recent military research and development. Since single-processor simulators do not support adequately large numbers of nodes, it would be quite advantageous to parallelize a simulator.

The traditional internet has arguably solved the problems of network discovery and routing for computers that are basically plugged into the wall and never moved. In these cases, network geometry does not change, with the exception of adding or removing nodes and/or links. Thus, shortest-path determinations remain valid once they have been found. In contrast, a MANET must be assumed to be constantly moving and reconfiguring (this assumption can be modified somewhat by considering the direction and general tendencies of node movement, but this topic is beyond the scope of this project). Thus, shortest-path routing must constantly be re-evaluated.

However, mobile ad-hoc networks have additional constraints in that they use wireless links. Whereas the traditional internet uses a physical wire to connect two computers separated by any distance (an admittedly slightly oversimplified model), a MANET can only broadcast its message to all nodes within a finite range. This has both advantages (a node can broadcast a UDP message to many nodes simultaneously) and disadvantages (multi-hop routing becomes more important) in terms of protocols. More important, however, is the ramifications of the wireless channel: this channel is noisy, subject to interference and blockage, and far easier to interrupt than a clean, shielded cable connection. In addition, MANET nodes are portable, and therefore run on batteries. Radio power must be conserved as much as possible, yet broadcast range requires increased power in order to reach farther. Similarly, network throughput is maximized when the network routing is at its most effective configuration; however, keeping track of network topology can easily consume all available bandwidth. A MANET is constantly juggling these sorts of tradeoffs; hence the driving need for a simulator that would permit experimentation with these parameters, especially on large numbers of nodes.

### **Literature Survey (Existing Solutions)**

Our research survey was based on results from Google, NEC Citeseer, and pointers from Dr. Baden.

Network simulators are often classified under the category of Parallel Discrete Event Simulators (PDES), especially for the specific variant of “spatially explicit problems”. PDES (and their use as simulators of traditional internet-style networks) have been around since 1970’s [5]. However, MANET simulators have only started to appear in the late 1990’s. Part of the reason for this is that the move from fixed, hardwired networks to MANETs has only recently been enabled by deployable technology. Another reason is that the type of problem is so difficult.

Networks have many different classifications. The easiest way to illustrate the role of MANETs in this family is by diagramming as follows:

1. internet (wired -- traditional)
2. internet (wireless)
  - a. infrastructure-based (e.g. cell phones; a WLAN based on access points)

- b. ad-hoc
  - i. immobile (e.g. land-based sensor networks)
  - ii. mobile (MANETs, e.g. air- or water-borne sensor nets)

Each level imposes additional complications to those of the level above it. As such, network simulators for the simpler levels are insufficient for the deeper levels. Our project focuses on Level 2.b.ii, for which we found limited products, especially in terms of parallelized simulators.

Parallel Discrete Event Simulators model a system as it evolves over time, with the caveat that system components only affect each other in discrete events (i.e. in messages or particles, and not in deformations or continuous forces, not counting the results quantum physics). Events become the currency of the simulation, and great care must be taken to ensure that all portions of the parallelized simulation (i.e. “timelines” or “logical processes” (LPs), which communicate over “channels”) stay synchronized, while still permitting independent processing.

Time synchronization is achieved with either of two approaches: conservative or optimistic [7]. The problem is ensuring that one LP does not receive a message at time  $(t+\delta t)$  when the message affected its state at time  $t$ . Conservative algorithms prevent late message arrival by using frequent barrier synchronizations, or by using lookahead. In contrast, an optimistic algorithm proceeds without the frequent synchronization checks, and when a late message arrives, it rolls back the simulation to the necessary time point. “Lazy rollback” refers to not backtracking the simulation if the late message wouldn’t have affected the current state if it had arrived on time. Conservative researchers are always trying to find more efficient ways of doing lookahead, while optimistic research focuses on reducing the cost of rollback.

In spatially explicit problems, the simulation space is partitioned and distributed among the processors. While multi-dimensional decomposition would permit better load balancing, single-dimensional decomposition is typically developed first because the software is far simpler [7]. Traditional network simulators do not bother with spatial representation, due to the relative irrelevancy of the spatial geometries of wired networks. However, mobile networks (both ad-hoc and infrastructure-based) certainly require spatial awareness, since their communication range is a finite. Thus, instead of the usual software architecture of functional or arbitrarily-partitioned nodal parallelization, the problem is divided spatially.

The most relevant PDES for our problem is the Dartmouth Scalable Simulation Framework (DaSSF) kernel developed by Jason Liu and David Nicol as an extension to the SSF API [19]. The DaSSF 3.0 kernel relies on the conservative lookahead strategy and supports both synchronous and asynchronous simulation protocols. It also supports distributed memory architectures, with processor synchronization performed in shared memory. Each compute node, and each processor within that node, is treated as a DaSSF subsystem (similar to PDNS), with communication done through standard MPI. Spatial partitioning is automatic from the user’s perspective. The kernel attempts to adjust

channels and communication protocols between subsystems to optimize for the model being simulated. [10]

UCLA's GloMoSim was another attractive candidate [21]. It is designed to support topology, routing, and other specific protocols at separate OSI-type levels, permitting easy experimentation with alternatives. The code supports a custom Event-driven API in C++, but requires Parsec compilation for certain special features. In most respects, it is similar to the other PDES already discussed.

There are also network simulators that are not based explicitly on PDES. These simulators tend to be for single serial computers (e.g. OPNET runs on Win32 machines) and are more commercial than academic; as a result, the code is not open for student development. The approach in serial simulators follows many of the design decisions of the PDES work, however the emphasis here is on multithreading. This area can be discussed in more detail on request.

**Justification** for our approach (no time to figure out their stuff + do enough work)

Our approach consisted of creating our own simplified parallelized MANET simulator, based on our own ideas and those of existing simulators. This decision was reached after first exploring the other possibilities. We first looked at non-PDES simulators. The most viable option here was simply using PDNS: the PADS research group at Georgia Tech has been working on parallelizing ns-2; however their approach attempts to minimize modifications to ns-2 [8]. Their strategy involves simply breaking the field of nodes into several individually running single-processor ns-2 simulations that coordinate periodically. This approach is difficult to modify such that nodes can move freely around the simulation space, and therefore we decided not to load this software onto valkyrie

Another option, namely parallelizing one of the other traditional serial-computer MANET simulators, was also unattractive, since OPNET requires heavy licensing fees (in the tens of kilobucks) and is not open source. Also, the authors are unfamiliar enough with multithreading to feel up to the challenge of parallelizing such code.

In the realm of parallel discrete event simulators (PDES), we found that there are, in fact, several existing parallelized network simulators available for download. Acting on this would have raised two questions, namely (1) how much time and effort would it take to download the code to valkyrie, learn the code, and get it working, given the history of problems with valkyrie in the course to date, and (2) what amount of modifications to the simulator would we then do to justify a course project. We decided that these two risks were too great, and thus confirmed our earlier decision to use existing simulators as guidance for writing our own simple simulator.

The Simulator for Wireless Ad-Hoc Networks (SWAN), based on the DaSSF kernel, was our favorite existing solution, as it was already designed for MANETs. However, it was infeasible mostly because it was discovered too late in the project. For example, the authors of SWAN took several months to implement the 802.11 protocols. [11]

Similarly, the existing non-MANET PDES simulators were insufficient for modeling MANETs without unacceptably heavy modifications.

GloMoSim claims to support rapid prototyping, and code investigated by the authors seems to verify this claim [21]. However, getting GloMoSim to run on a single-processor laptop proved difficult, and due to the nature of the author's student funding sources, the ability to work from an off-network Linux laptop was essential to this project. Also, GloMoSim requires the use of the Parsec language, and frankly the project appears somewhat desecoped by UCLA, since there has been a lack of updates since early 2001 (as with PDNS... there must have been a conference in February 2001).

We also briefly looked at the IP-TNE extension to the Taskit kernel [17]. Unfortunately this software is meant for distributed grid-computing machines, not a parallel cluster like valkyrie. In addition, it had potential bottlenecks in that a single emulator process is inserted between *all* network messages, for the purpose of packet re-addressing and aliasing. Lastly, IP-TNE is meant to simulate real-time network latency for the purposes of testing multiplayer video games. While we obtained some useful insights from this work, we felt it was unwise to pursue it at the development level.

We therefore decided to build our own relatively simple simulation software.

### Our **Approach** (high level)

Our simulator starts with a randomly distributed set of MANET nodes. Network discovery protocols can be based on standard, well-developed commercial protocols, or manually coded experimental protocols. Our work is in analyzing the effects of moving nodes, testing various content-based distribution protocols, and investigating the analogous behavior of a parallel machine and a network. We are primarily interested in the scaling and overhead requirements of the protocols.

The parallelization of such a simulator is the primary task. MANET nodes could be distributed around the parallel computer's nodes. Ideally, nodes could directly implement the experimental routing protocols, and "simulate" passing messages between mobile nodes by actually using that protocol to send across the nodes of the parallel machine.

The main tasks of the software were:

1. Establish an initialization process where a set of  $n$  mobile nodes are distributed across  $P$  processes.
2. Either run a network discovery algorithm in parallel, or assume the mobile nodes already have that information.
3. Simulate and test various content-based routing protocols in parallel, as nodes move around the simulated geographic space. Balance simulation load as mobile nodes clump or network traffic becomes heavier in localized regions.

### Our **Architecture** (detailed)

Our main compute platform, valkyrie.ucsd.edu, is running the [Rocks](#) system developed at the [San Diego Supercomputer Center](#). Valkyrie consists of 16 dual 1GHz Pentium III CPUs. Each node has 1GB of RAM, and a Myrinet switch provides low latency connectivity between the nodes [18]. We use the MPI 1.1 implementation for message-passing [29].

In this report, we will use the following terminology to avoid the naming confusion inherent in using one cluster (valkyrie) to simulate other clusters (the MANETs).

- Node = simulated MANET node
- Proc = Beowulf compute node (processor and/or process)
- $r$  = length of one side of a cell. A partition is spatially subdivided into cells.
- $R$  = radio broadcast radius, in units of  $r$

Any model must start with a set of assumptions, and ours is no different.

1. Message packets are always completely sent/received
2. One packet is sent per timestep
3. There are no packet collisions
4. Broadcast geometry is a square

Assumptions (1) and (3) are acceptable, because we apply a percentage-loss factor to the nodes attempting to receive a message. This factor can easily accommodate the two assumptions. The second “assumption” is really just our definition of “timestep”, but is included under “assumptions” for simplicity. Assumption (4), the square-shaped area-of-effect for each radio, can be justified by noting that the area of a square ( $4r^2$ ) is larger than the area of a circle ( $\pi r^2$ ) only by a factor of roughly 1.27. We argue that this is close enough to unity, especially since a circle is also only an approximation of radio propagation from an omnidirectional antenna in free space. As Figure 1 shows, real-world broadcast patterns exhibit roughly circular shapes on average, but a square is also a reasonable approximation [9]. Most spatially-oriented simulators reviewed in the Literature Survey used either squares or hexagons.

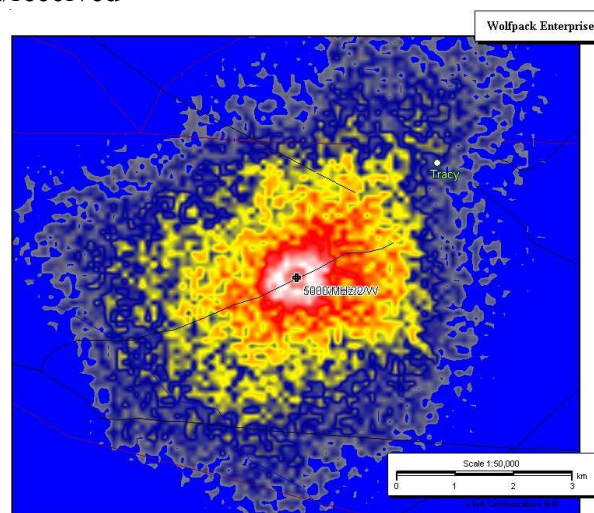


Figure 1: 2.4 GHz Broadcast Pattern

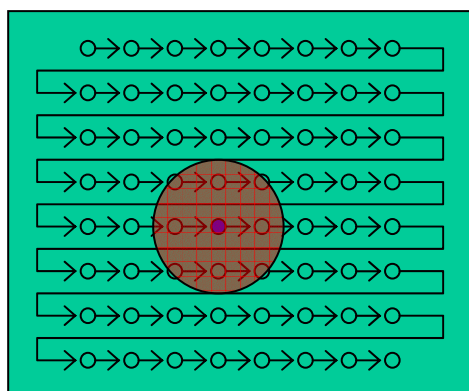


Figure 2: Original Data Structure

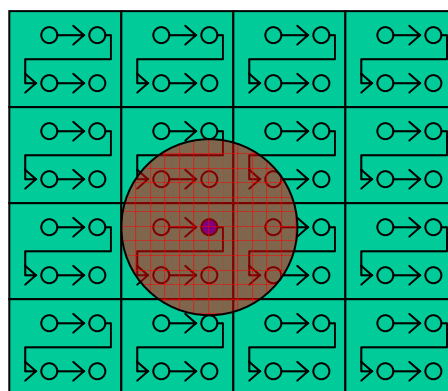


Figure 3: New Data Structure

We initially thought of having a doubly linked list of network nodes for each processor, in order to enable easy movement of MANET nodes from processor to processor (this data structure allowed node representations to be deleted from one processor's node list and added to another in unit time, once the node crossed from one processor's domain to another).

Unfortunately this resulted in unacceptable run times due to the distance calculations. Since we are simulating wireless networks, the node has a limited area of effect for broadcasting a message. To accurately simulate the network, each time a node sends a packet, all its neighbors must be notified of transmission. However, because our first attempt at a data structure (the doubly linked list) had no notion of spatial locality, each node had to measure its distance in simulation space from every other node to determine if a pairwise effect existed. This is shown in Figures 2 and 3, with green circles representing nodes, arrows showing linked list connections, the blue node indicating a node that is currently transmitting, and the red circle showing its area of effect.

To remedy this, we changed the worldview to impose a matrix of geographically oriented cells on the simulation space. Each cell can hold multiple MANET nodes. Now, node areas of effect are far simpler to calculate, since we need look only at nodes within affected cells, and we can easily calculate how many cells are affected by looking at the position of the transmitting node and the communication radius. This also greatly simplifies processor partition boundaries, since we can just use message centers of mass in the ghost cells.

Our initial design goal was to have each processor be capable of handling 10,000 MANET nodes. Unfortunately, on a typical valkyrie processor, the time required to run a Pythagorean distance measurement from each node to each node was on the order of 50 seconds, much too long for a single iteration. Under the new data structure, because we must check between 1/5 and 1/10 as many nodes (to see if nodes in affected cells are actually within the circular radius of communication), we see decreases in running time of between one and two orders of magnitude, to an average of 0.3 seconds. If we ultimately decide to decrease cell sizes sufficiently to assume that any cell touching the communication radius may be assumed to also have all its nodes within the area of effect, the step of figuring out what nodes are affected becomes essentially constant-time.

Simulation space was partitioned into vertical strips. This is one of several of our architectural ideas that were happily confirmed by Deelman and Szymanski's spatially explicit PDES simulations of disease spreading [7]. 1-dimensional partitioning was implemented first, since the code is far simpler than other partitioning strategies, since there are fewer shared borders. However, an interesting non-contiguous partition placement quirk was developed as a consequence of our "wrapping" strategy, below.

The next major architectural point is the treating the simulation space as an infinite plane. Much like the arcade game “Asteroids”, a node that wanders off the edge of the simulation terrain will appear to enter at the opposite edge (see Figure 4) [2].

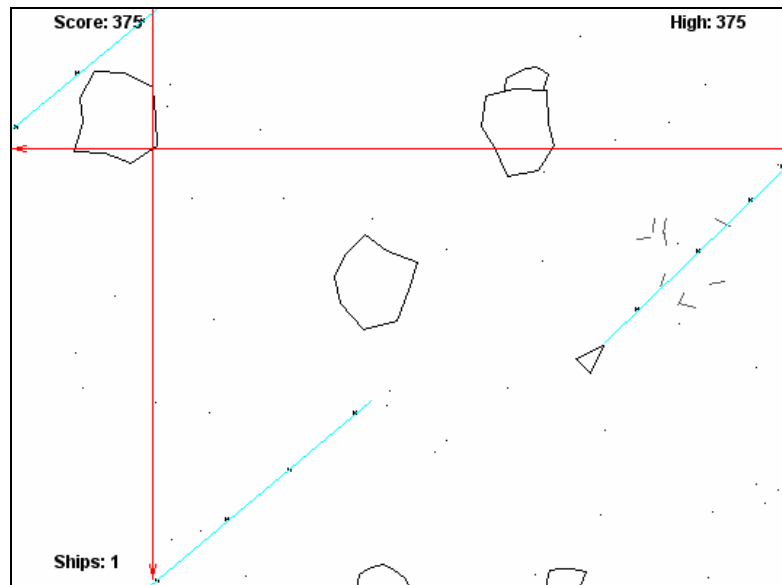


Figure 4. Asteroids screen shot, with spatial wrapping (right to left, followed by top to bottom) augmented for clarity

This approach offers several advantages; namely, we can guarantee that the desired number of nodes will participate in the entire simulation run, and we can write code that easily adapts to almost any partition geometry, including irregular meshes and even the single-processor case. In theory, we can also arrange partitions asymmetrically, as in Figure 6.

Then, for a single square region of the framework, we can approximate an infinite area by connecting the top/bottom and left/right edges together to form a torus as seen in Figure 5 on the right.

Wrapping node movement permitted us to easily write and test code on a single processor. Also, wrapping lends itself to arbitrary spatial partition arrangements across the parallel machine.

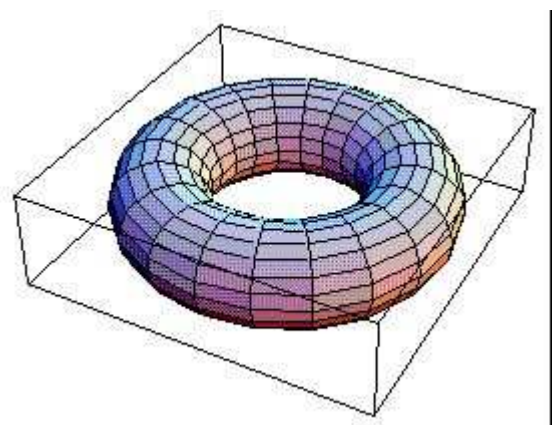


Figure 5: Model of an Isolated Simulation Frame

This does introduce some complexities, such as how to wrap messages across borders and across corners. The innovative pointer-based structure, described later, of our simulation space enabled us to do this very efficiently.

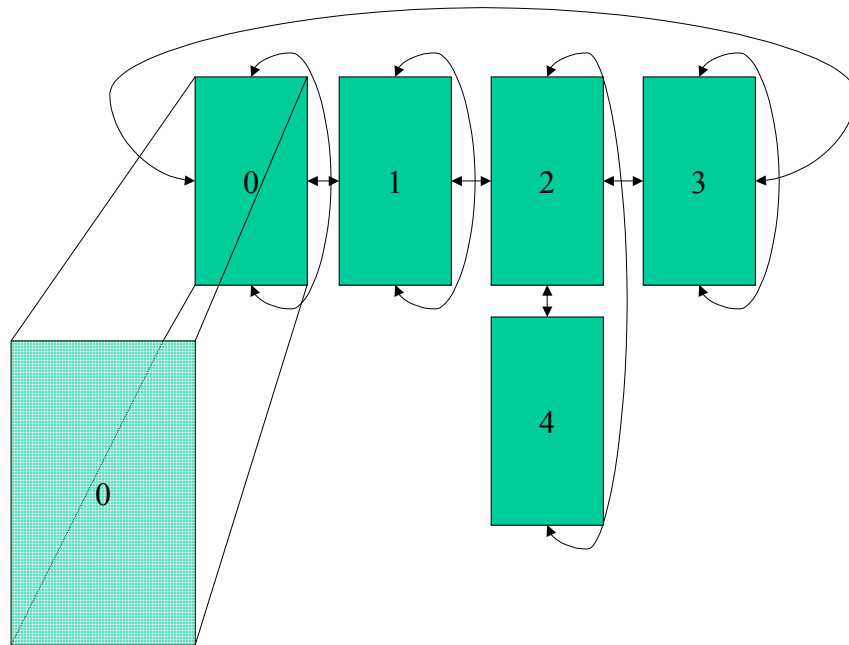


Figure 5. Adaptive Partition Placement Enabled by Spatial Wrapping

Nodes are basic classes (see Appendix A) that can be subclassed by MANET network protocol. This enables us to use the same simulation code, and simply inherit the basic node code into a class that supports the Send and Receive functions we wish to evaluate. For this project, we planned mostly on protocols of basic wide-flooding broadcasts of small, periodic location messages (e.g. GPS). In the future, we would also like to investigate the costs of routing (e.g. directed email or voice-over-IP multicast).

One method of network discovery and topology we modeled after the ODMRP work done at UCLA PCL and its graduates working at Sensoria [3, 6, 22]. A simple timing algorithm determines if a node is just a member of a node cluster, or if it is the clusterhead that maintains cluster membership details. Under our code design, it was straightforward to model the frequency of membership messages and maintenance versus responsiveness to changes in network topology as the nodes move.

Node movement is accomplished by random walks. We are also implementing programmable movement patterns such as sweep in a certain direction, wander in a circle, and march towards/away from spatial center. We can set movement to zero, thus simulating a fixed wireless network. Simulating a wired network would require substantial changes to the code due to the differences in signal propagation. The benefit of laboring over these changes would perhaps be negligible, considering the excellent simulators that already exist for these types of networks.

Initial node dispersion was based on a random number generator. The simulator places nodes in a random dispersion for the initial distribution of nodes, for efficiency. Note that the code can easily be modified to have all node initialization coordinated by the individual node, although this comes at some cost as the simulation size increases. For experimental validity, it was important to have the randomized node placement be repeatable for arbitrary experiments.

For the simulated messaging, nodes passed packets to each other. The packets were a simple C++ class that simulated a GPS location message, a network topology information packet, a routing information packet, or whatever we wished to simulate.

Nodes placed their Sent packets into the Receive queues of their neighboring nodes, within broadcast range. This was optimized using a spiral walk to the upper left corner, and then simultaneously walking from left to right, in the same direction as memory was accessed on valkyrie. At the next synchronization point in the timestep, each node pulled the messages from their home cell's Receive queue only. This Send and Receive process was simplified using the home cells as the abstraction of communication, which greatly reduced parallel communication costs and memory access times.

Ghost cells are used to pass node messages and, in the case of node movement and load balancing, nodes themselves across partition boundaries. The messages are passed during the messaging portion of the timestep, and the nodes are passed during the node movement re-positioning portion. This was non-optimal in that it required two phases of inter-processor communication, however this was unavoidable due to the decision to wrap node movement but not node messaging. That being said, inter-processor communication was otherwise kept to a minimum

### **Architectural Stuff We Thought Was Way Cool at 4 AM**

The simulation "framework" on each processor for keeping track of the cells was organized using a 2D doubly linked list. This had many advantages, in both efficiency and reduction of complexity. When moving nodes or broadcasting messages within a framework, we can simply follow pre-established links to determine where the objects should be passed, instead of iteratively querying cells to find out if they are border cases.

Load balancing was accomplished by allowing frameworks to peel off 1D "stripes" of cells containing nodes/packets and pass them to a neighboring processor. Frameworks automatically attempt to equalize the number of nodes between them and their neighbors, keeping each processor with as much work as possible. The very tangible benefits of enabling load sharing based on node number are shown in Figure 6-8.

To keep communication latency to a minimum, we have established a fixed pattern of communication between odd and even numbered nodes that allows everything to be done with MPI\_Send and MPI\_Recv. This requires an even number of processors to take part in the simulation, but we do not view this as an important limitation. Necessary global

information is either propagated slowly throughout the entire simulation space, or is projected from information obtained by neighboring frameworks.

Packets are modeled as physical entities; they can contain real data and direct nodes in movement or protocol selection. Because a single packet may be up to 64 kb, therefore, it was important to minimize the number of simultaneously existing instances of a particular packet. This was accomplished by propagating pointers to packets instead of the packets themselves. Whenever a modification had to be done to a packet (hopcount incremented, for instance), we simply split the packet into two different versions. However, this on-demand requirement resulted in much lower memory use than we otherwise would have experienced. A difficulty with this approach is efficiently keeping track of which packets are still in use and which are not. The solution given uses automatic garbage collection, in the sense that references to packets are kept track of, and the packet is deleted as soon as it looks like no references remain.

## Our **Results**

The simulator has some quite pleasing results. The application naturally lends itself to a parallel solution, because much of the work can be done internally and external communication may be quite limited in many cases. It is worth mentioning, however, that processors are essentially “locked” together by the constraint that no processor may finish an iteration before its neighbors. Hence, if the distribution of nodes (load) is very uneven, we will achieve no speedup over the serial case.

This is the motivation behind our focus on load-balancing, and as may be seen in Figure 6 below, the work has paid off. To generate Figure 6, we placed 1000 nodes on the even processors of a six-processor configuration, and no nodes on the odd processors. Nodes were told to move very slowly if at all to emphasize this worst case. As may be seen, by the end of only 50 timesteps, the load balancing version of the code had a performance improvement of almost 20% over the non-load balancing code. It is interesting that there is a crossover point in the graph. This is due to the fact that relatively few packets had been transmitted before the 20<sup>th</sup> step of the simulation, and hence simulation was simply traversal of the framework. After the 20<sup>th</sup> step, however, the network began to see significant throughput, resulting in complexity that could be effectively shared among processors.

Fig 6: Effect of Load Balancing w/ 6 Processors, Initial Node Placement on Even Processors

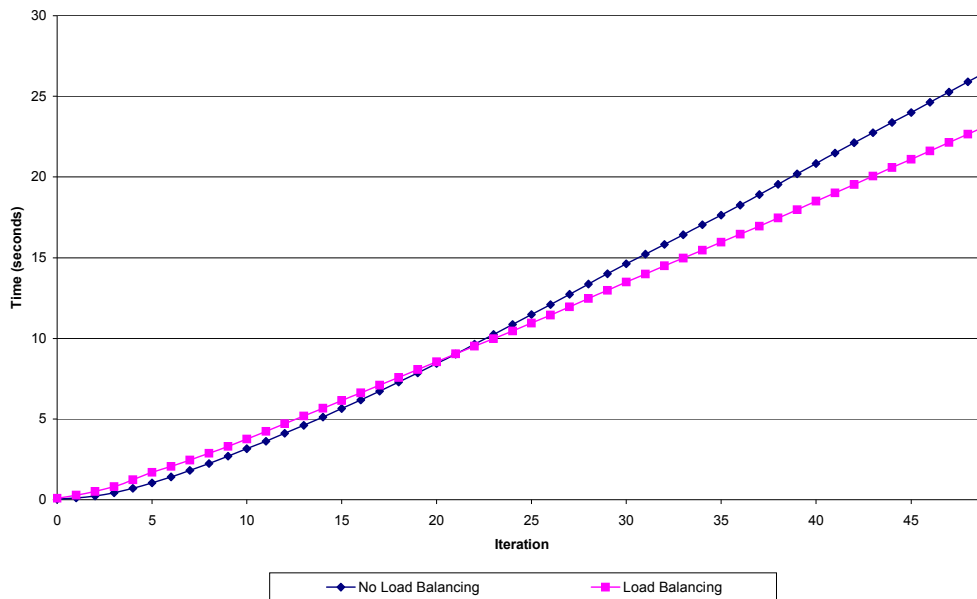


Figure 7, below, shows an even more extreme case. This occurs when all nodes have accumulated within a very constrained region of space. In our example scenario, 6000 nodes are within one processor's domain, out of a possible 6 domains. Here, the crossover point comes much earlier, due to the much higher number of nodes (and therefore the equivalently probability of a transmitted packet). By 50 iterations, the non load-balanced version is taking 1.5 times as long as the load-balanced version.

Fig 7: Effect of Load Balancing w/ 6 Processors, Initial Node Placement on Single Processor

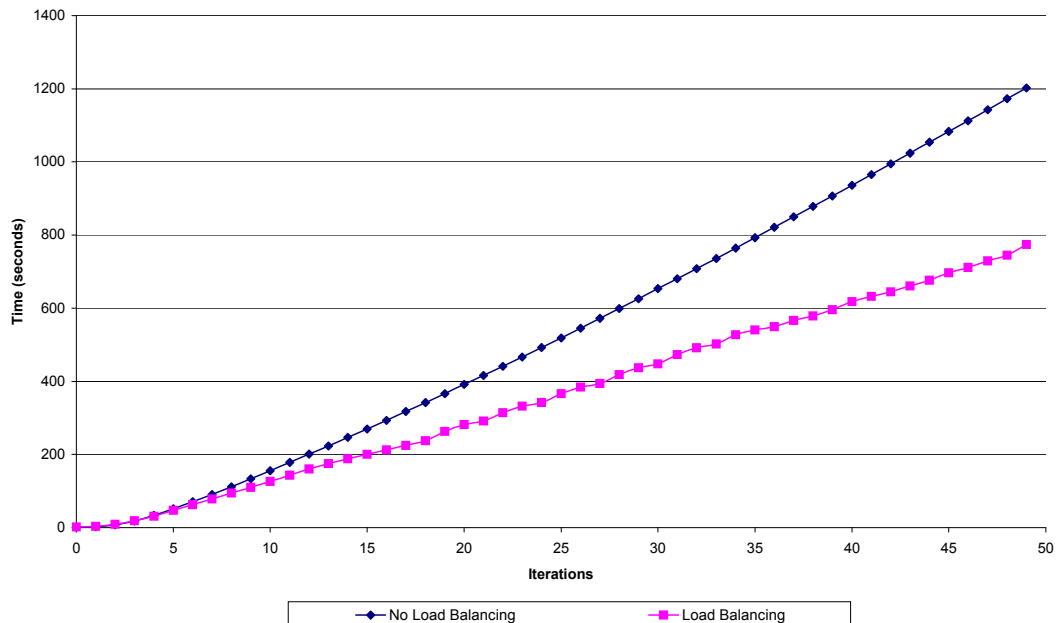
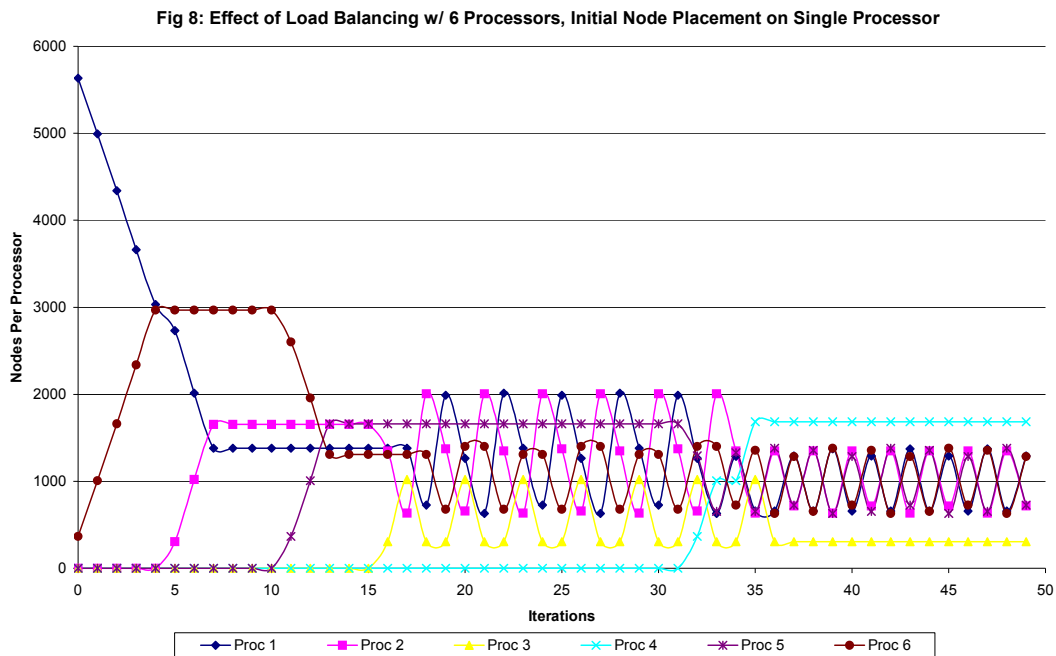
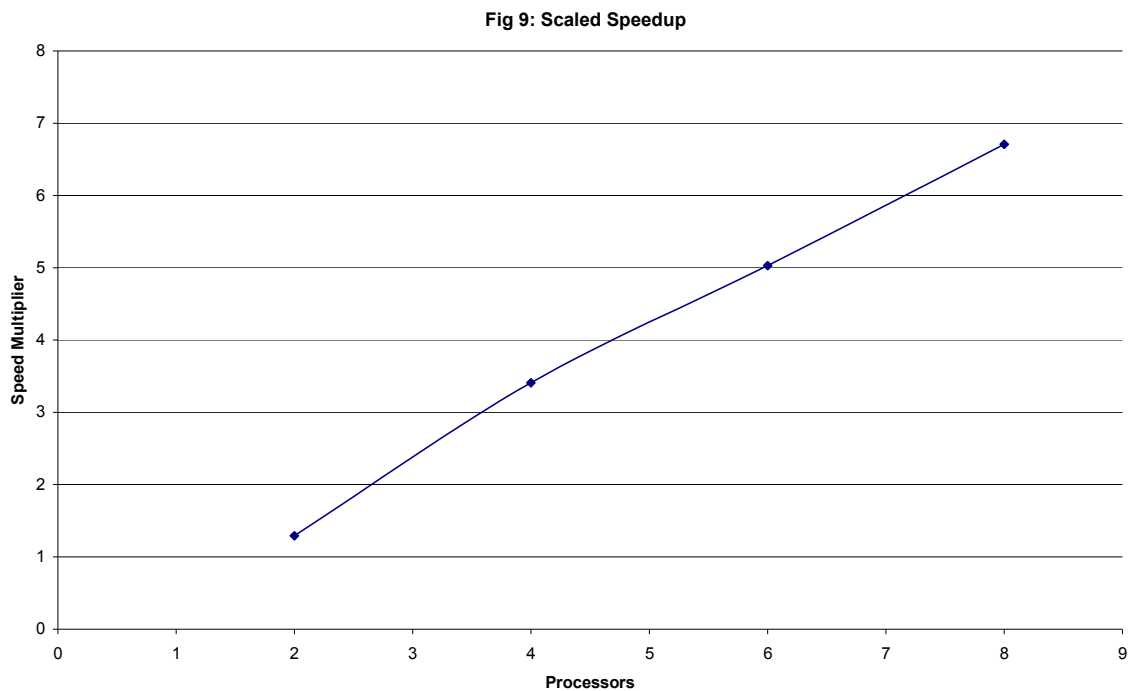


Figure 8, below, shows a plot of the number of nodes simulated by each processor in the above load-sharing example. As may be seen, Processor 1 begins with 6000 nodes in timestep 0. The connected Processors, 6 and 2, are the first to see and respond to the load imbalance. Processor 6 is first because of the way the code is written. To avoid sending too much data, no processor can send more than 1 column at a time. Higher-numbered processors have greater priority, and hence Processor 6 is first to share the load. When 6 and 1 have equalized, 2 further diminishes 1's load. These secondary processors then share load with tertiary processors farther back in line, and so forth.

The network appears to be in stasis by approximately iteration 35. The sinusoidal behavior seen as the processors attempt to equalize load may be reduced by increasing the number of vertical slices appearing in a framework. This will allow for finer granularity (though also a slower stabilization), leading to a more quiescent solution.



The scaled speedup of the load-balancing algorithm over the serial version is approximately linear with the number of processors. This is expected when the load-balancing code is working properly.



## Correctness

After experimenting with a number of mobile simulators, we've concluded that we can't perform meaningful comparisons with any of them. The primary reason is that the tuning parameters of our simulator are different than many of theirs. Our simulator is intended to describe the interactions and traffic flow of many thousands of nodes, of which the precise position is not of primary importance. Other parameters, such as node state, probability of transmitting, probability of interference, and movement are treated at a very high level. One can specify models for each of these attributes, but it is difficult to implement a particular pattern for a subset of nodes without changing them all.

To verify correctness, we tested the simulator against the most difficult cases of the protocols implemented, but only on a small scale. To ensure that nodes acted as "repeaters," we set up a single chain of nodes across frameworks, had them move with random Brownian motion, and ensure that the only packets left unpropagated were dropped due to physical limitations such as distance.

## Conclusions

If we had to do it over again, we would have chosen to download SWAN/DaSSF and work on extending it. However, we learned a lot about parallel programming and discrete event simulation in the project, and picked up a little about wireless networking along the way. Having a tangible purpose for the assignment was a huge motivator. There has been a significant lack of work done in this field, and our project could easily

be extended into publishable research. We spent the majority of this project learning the field and making the simulator, and did not have as much time as we would have liked to actually use the simulator to obtain useful networking results.

### **Further Work**

- Further experiments to test these and other factors in greater detail.

- Initial node distribution via an input file, not just randomization. This would permit evaluation of specific network geometries.

- Support for additional routing protocols.

- Support for additional network discovery algorithms.

- Adaptive mesh refinement or other advanced load-balancing strategies.

## Bibliography

1. Allene, P. and Tropper, C. On the parallel simulation of fixed channel allocation algorithms. *Mobile Networks & Applications* (5.3): 209-218 (2000).
2. Asteroids screen credit: <http://javaboutique.internet.com/asteroids/>, copyright 1998 Mike Hall
3. Bajaj, L., M. Takai, R. Ahuja, K. Tang, R. Bagrodia, and M. Gerla. 1999. Glomosim: A scalable network simulation environment. Technical Report 990027, UCLA, Computer Science Department.
4. Carl Tropper. Parallel Discrete Event Simulation-Applications. Unknown source, referred to us by Dr. Scott Baden, UCSD. [http://www-cse.ucsd.edu/users/baden/classes/cse260\\_fa02/Projects/PDES/](http://www-cse.ucsd.edu/users/baden/classes/cse260_fa02/Projects/PDES/).
5. David Nicol and Jason Liu. Composite Synchronization in Parallel Discrete Event Simulation. *IEEE Transactions on Parallel and Distributed Systems* (13.5): 433-446 (May 2002).
6. Ed Cheng, [On-demand Multicast Routing in Mobile Ad Hoc Networks](#), Carleton University, Ontario, Canada, January 2001.
7. [Ewa Deelman](#), Boleslaw K. Szymanski: Simulating Spatially Explicit Problems on High Performance Architectures. *Journal of Parallel and Distributed Computing* 62(3): 446-467 (2002)
8. Georgia Tech PADS Research Group, <http://www.cc.gatech.edu/computing/compass/pdns/>, November 2002 (last updated February 2001).
9. Green Bay Professional Packet Radio, [Wireless Link Analysis Software](#), November 2002.
10. Jason Liu, DaSSF, <http://www.cs.dartmouth.edu/~jasonliu/projects/sf>, November 2002.
11. Jason Liu, David M. Nicol, L. Felipe Perrone, and Michael Liljenstam. Towards high performance modeling of the 802.11 wireless protocol. In *Proceedings of the 2001 Winter Simulation Conference (WSC 2001)*, December 9-12, 2001, Arlington, VA.
12. Jason Liu, L. Felipe Perrone, David M. Nicol, Michael Liljenstam, Chip Elliott, and David Pearson. Simulation Modeling of Large-Scale Ad-hoc Sensor Networks. *European Simulation Interoperability Workshop 2001*.
13. ns-2 Working Group, <http://www.isi.edu/nsnam/ns/>, November 2002.
14. [On-Demand Multicast Routing Protocol](#), S.-J. Lee, M. Gerla, and C.-C. Chiang, *Proceedings of IEEE WCNC'99*, New Orleans, LA, Sep. 1999, pp. 1298-1302.
15. OPNET Technologies, Inc., <http://www.opnet.com/>, November 2002.
16. P. S. Pacheco, "Parallel Programming with MPI", San Francisco: Morgan Kaufmann Publishers, Inc, 1997.
17. Rob Simmonds, Russell Bradford, and Brain Unger: Applying Parallel Discrete Event Simulation to Network Emulation. *14th Workshop on Parallel and Distributed Simulation (PADS 2000)*: Bologna, Italy, May 28 - 31, 2000
18. Scott Baden and Steven Lau. [http://www.cse.ucsd.edu/users/baden/classes/cse260\\_fa02/testbeds.html](http://www.cse.ucsd.edu/users/baden/classes/cse260_fa02/testbeds.html) as of December 6, 2002.
19. SSF Research Network, <http://www.ssfnet.org/homePage.html>, November 2002.

20. Theodore V. Hromadka III and Naomi S. A. Ramos. Mobile Ad-Hoc Network Topology Optimizations. Student project, UCSD CSE 202 Fall 2001.
21. UCLA Parallel Computing Laboratory, <http://pcl.cs.ucla.edu/projects/glomosim/>, November 2002 (last updated February 2001).
22. [Unicast Performance Analysis of the ODMRP in a Mobile Ad hoc Network Testbed](#), S.H. Bae, S.-J. Lee, and M. Gerla, *Proceedings of IEEE ICCCN 2000*, Las Vegas, NV, Oct. 2000.
23. William Gropp and Ewing Lusk <http://www-unix.mcs.anl.gov/mpi/mpich/>

## **Acronyms**

IP-TNE – Internet Protocol Traffic and Network Emulator

LP – Logical Process

NS – Network Simulator

MANET – Mobile Ad-Hoc Network

ODMRP – On-Demand Multicast Routing Protocol

PADS – Parallel and Distributed Simulation

PDES – Parallel Discrete Event Simulation

PDNS – Parallel/Distributed NS

## **Appendix A: Code**

### **Main Application:**

```
#include "node.h"

using namespace parallel;

#define NODES      6000
#define ITS 50
#define RADIUS     .5

void main(int argc, char **argv) {

    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int cells=10, mps = 10, err;
    Framework f1(cells, mps, size, rank);

    for (int i = 0; i < NODES; i++)
        f1.AddNode(new Node(i+NODES*rank,
random()%(cells*mps), random()%(cells*mps), frand()));

    for (int i = 0; i < ITS; i++) {
        f1.Iterate();
        cout.flush();
    }

    cout.flush();
    MPI_Finalize();
}
```

## Header File:

```
#include <set>
#include <mpi.h>

namespace parallel {

    // #define lfloat double
    #define lfloat unsigned int
    #define byte unsigned char
    #define MAXHOPCOUNT 100
    #define MAXNODES 10000
    #define COMMRADIUS 1
    // #define DEBUG
    #define frand() (float) random()/RAND_MAX
    // #define frand() random()

    #define MPINODETYPE 12
    #define MPIPACKETTYPE 13
    #define MPICELLTYPE 14
    #define MPINODECOUNTTYPE 15
    #define MPILBTYPE 16

    enum PACKETTYPE
    {PING=1*sizeof(int),ACK=1*sizeof(int),JOIN=1*sizeof(int),DATAS=64*sizeof(int),DATAM=512*sizeof(int),DATAL=65536*sizeof(int)};

    enum BORDERTYPE {NONE=0, LEFT, RIGHT};

    struct pos {
        lfloat x,y;
    };

    struct packwrapstruct {
        int source, dest, starttime, type, seqno, hopcount, currtime,
        rowid, over, up, dir, alive;
    };

    struct nodewrapstruct {
        int myid, pacrec, seqno, posx, posy, txprob, rowid;
        int history[MAXNODES];
    };

    class Packet;

    class PacketWrapper {
    public:
        PacketWrapper(int source, int dest, int starttime, int type, int
        seqno, int hopcount, int currtime, bool alive, int rowid, int over, int
        up, int dir);
        PacketWrapper(Packet* p, int rowid, int over, int up, int dir);
        PacketWrapper(packwrapstruct* pws);
        packwrapstruct* GetPWS();
        void Print();
    private:
```

```

        void Initialize(int source, int dest, int starttime, int type,
int seqno, int hopcount, int currrtime, bool alive, int rowid, int over,
int up, int dir);
        packwrapstruct pws;
};

class Packet {
public:
    Packet();
    Packet(int source, int dest, int starttime, int type, int seqno);
    Packet(int source, int dest, int starttime, int type, int seqno,
int hopcount, int currrtime, bool alive);
    Packet(packwrapstruct* pw);
    ~Packet();

    Packet* Clone();

    void SetHopCount(int hopcount);
    int GetHopCount();

    void SetAlive(bool alive);
    bool GetAlive();

    void SetCurrTime(int time);
    int GetCurrTime();

    int GetStartTime();
    int GetSource();
    int GetDest();
    int GetType();
    int GetSeqNo();
    int* GetMessage();

    void Print();
private:
    void Initialize(int source, int dest, int starttime, int type,
int seqno);
    int source, dest, hopcount, seqno, starttime, currrtime, alive;
    int* message;
};

class Node;

class NodeWrapper {
public:
    NodeWrapper(Node* n, int rowid);
    NodeWrapper(nodewrapstruct* nws);
    nodewrapstruct* GetNWS();
    void Print();
private:
    void Initialize(int myid, int pacrec, int seqno, int posx, int
posy, int txprob, int* history, int rowid);
    nodewrapstruct nws;
};

class Node {

```

```

public:
    Node(int myid);
    Node(int myid, lfloat x, lfloat y, float txprob);
    Node(nodewrapstruct* nws);
    ~Node();

    void SetPos(lfloat x, lfloat y);
    pos GetPos();

    int GetID();
    float GetTxProb();
    int GetPacketsReceived();
    int* GetHistory();
    int GetSeqNo();

    Packet* TxPacket(int dest, PACKETTYPE type);
    Packet* RxPacket(Packet* p);

    bool MoveRandom(int cellsize);
    void WrapPos(int cellsize);

    void Print();
private:
    void Initialize(int myid, lfloat x, lfloat y, float ltxprob);
    pos p;
    int myid, pacrec, seqno;
    float txprob;
    int* history;
};

class Cell {
public:
    Cell(int size);
    ~Cell();
    void AddNode(Node* ptr);
    void AddPacket(Packet* ptr);
    Packet* GetPacket();
    void TxPackets(int dest, PACKETTYPE type);
    void RxPacket(Packet* p);
    void RxAll();
    int GetCount();
    int GetSize();

    void Print(int i, int j);
    void Print();
    void SetBorder(byte value);
    byte GetBorder();

    void MoveNodes(int row, set<NodeWrapper*>*
nleft, set<NodeWrapper*>* nright);
    void GetData(int row, set<PacketWrapper*>* pw, set<NodeWrapper*>*
nw);

    Cell* top, *bottom, *left, *right;
    set<Packet*> outgoing;
    set<Packet*> incoming;
private:

```

```

        void PrintList();
        int size;
        set<Node*> nodelist;
        byte isborder;
};

class Framework {
public:
    Framework(int dimension, int meterspercell, int size, int rank);
    ~Framework();
    void AddNode(Node* ptr);
    void TxPackets(int dest, PACKETTYPE type);
    void RxPackets();
    void MoveNodesInternal();
    void MoveNodesExternal();
    void Iterate();
    Cell* GetCell(lfloat x, lfloat y);
    Cell* GetGeoCell(int x, int y);
    Cell* GetCell(float x, float y);
    int GetDimension();
    void LoadBalance();
    int GetTimeStep();
    bool DoesCrossFrames(Cell* c, int row);
    void Print();
private:
    void RemoveCol(set<PacketWrapper*>* pw, set<NodeWrapper*>* nw,
Cell* ptr);
    void InsertCol(set<PacketWrapper*>* pw, set<NodeWrapper*>* nw,
Cell* ptr);
    int dimension, scale, nodecount, whichframe, nprocs, err;
    void PropagateInternal(Cell* c);
    void PropagateInternal();
    void PropagateExternal();
    void PropagateInternal(Packet* p, int startrow, int horiz, int
dir);
    void MPISendPW(set<PacketWrapper*>* myset, int dest);
    void MPIGetPW(set<PacketWrapper*>* myset, int source);
    void MPISendNW(set<NodeWrapper*>* myset, int dest);
    void MPIGetNW(set<NodeWrapper*>* myset, int source);
    void GarbageCollect();
    Cell** cells;
    Cell* head;
    set<PacketWrapper*> pleft;
    set<PacketWrapper*> pright;
    set<NodeWrapper*> nleft;
    set<NodeWrapper*> nright;
    double timeit, timetotal;
    MPI_Status status;
};

}

```

### Simulator Code:

```
#include "node.h"
#include <stdlib.h>
#include <iostream.h>
#include <math.h>

//#define DEBUG 1

#define COND (0)

namespace parallel {
int zork2;
set<Packet*> dellist;

#define roundf(x) ((x>=.5) ? 1 : 0)

/*****Packet Class*****/

Packet::Packet() {
    Initialize(0, 0, 0, 0, 1);
}

Packet::Packet(int source, int dest, int starttime, int type, int
seqno) {
    Initialize(source, dest, starttime, type, seqno);
}

Packet::Packet(int source, int dest, int starttime, int type, int
seqno, int hopcount, int currtime, bool alive) {
    Initialize(source, dest, starttime, type, seqno);
    SetHopCount(hopcount);
    SetCurrTime(currtime);
    SetAlive(alive);
}

Packet::Packet(packwrapstruct* pw) {
    Initialize(pw->source, pw->dest, pw->starttime, pw->type, pw-
>seqno);
    SetHopCount(pw->hopcount);
    SetCurrTime(pw->currtime);
    SetAlive(pw->alive);
}

Packet::~~Packet() {
    if (message != NULL) delete message;
}

void Packet::Initialize(int source, int dest, int starttime, int type,
int seqno) {
    this->source = source;
    this->dest = dest;
    this->hopcount = 0;
    this->starttime=zork2;
    this->seqno = seqno;
    this->alive=true;
    this->currtime = zork2;
}
```

```

        if (type == 0)
            message = NULL;
        else {
            message = new int[type];
            if (!message) {
                cout << "nullpointer returned from memory reference"
<< endl;
                exit(1);
            }
            message[0] = type;
        }
    }

void Packet::SetHopCount(int hopcount) {
    this->hopcount = hopcount;
}

int Packet::GetSource() {
    return source;
}

int Packet::GetDest() {
    return dest;
}

int Packet::GetHopCount() {
    return hopcount;
}

int Packet::GetStartTime() {
    return starttime;
}

int Packet::GetType() {
    return (message==NULL) ? 0 : message[0];
}

int Packet::GetSeqNo() {
    return seqno;
}

int* Packet::GetMessage() {
    return message;
}

void Packet::SetAlive(bool alive) {
    this->alive = alive;
}

bool Packet::GetAlive() {
    return alive;
}

void Packet::SetCurrTime(int time) {
    currtime =time;
}

```

```

int Packet::GetCurrTime() {
    return currtime;
}

Packet* Packet::Clone() {
    Packet* p = new Packet(source,dest,starttime,message[0],seqno);
    p->SetCurrTime(currtime);
    p->SetAlive(alive);
    p->SetHopCount(hopcount);
    return p;
}

void Packet::Print() {
    cout << "Packet: seqno " << seqno << ", hopcount " << hopcount
    << ", timestep " << zork2 << ", dest " << dest << ", source "
        << source;
    if (message != NULL) cout << ", type " << message[0] << endl;
    else cout << ", type is NULL" << endl;
}

/*****PacketWrapper Class*****/

PacketWrapper::PacketWrapper(    int source, int dest, int starttime,
int type, int seqno,
                                int hopcount, int currtime, bool alive,
int rowid, int over, int up, int dir) {
    Initialize(source, dest, starttime, type, seqno, hopcount,
currtime, alive, rowid, over, up, dir);
}

PacketWrapper::PacketWrapper(Packet* p, int rowid, int over, int up,
int dir) {
    Initialize( p->GetSource(), p->GetDest(), p->GetStartTime(), *(p-
>GetMessage()), p->GetSeqNo(), p->GetHopCount(),
                p->GetCurrTime(), p->GetAlive(), rowid, over, up,
dir);
}

PacketWrapper::PacketWrapper(packwrapstruct* pws) {
    memcpy(&(this->pws), pws, sizeof(packwrapstruct));
}

void PacketWrapper::Initialize(int source, int dest, int starttime, int
type, int seqno,
                                int hopcount, int currtime, bool alive,
int rowid, int over, int up, int dir) {
    pws.source = source;
    pws.dest=dest;
    pws.starttime=starttime;
    pws.type = type;
    pws.seqno = seqno;
    pws.hopcount = hopcount;
    pws.currtime = currtime;
    pws.alive = alive;
    pws.rowid = rowid;
}

```

```

        pws.over = over;
        pws.up = up;
        pws.dir = dir;
    }

packwrapstruct* PacketWrapper::GetPWS() {
    return &pws;
}

void PacketWrapper::Print() {
    cout << pws.source << " " << pws.dest << " " << pws.starttime
<< " " << pws.type << " " << pws.seqno << " " << pws.hopcount << " " <<
pws.currtime << " " << pws.alive << " " << pws.rowid << " " << pws.over
<< " " << pws.up << " " << pws.dir << endl;
}

/*****NodeWrapper Class*****/

NodeWrapper::NodeWrapper(Node* n, int rowid) {
    Initialize( n->GetID(), n->GetPacketsReceived(), n->GetSeqNo(),
(int) n->GetPos().x, (int) n->GetPos().y,
        (int) (n->GetTxProb()*RAND_MAX), n->GetHistory(),
rowid);
}

NodeWrapper::NodeWrapper(nodewrapstruct* nws) {
    Initialize(nws->myid, nws->pacrec, nws->seqno, nws->posx, nws->
posy, nws->txprob, nws->history, nws->rowid);
}

nodewrapstruct* NodeWrapper::GetNWS() {
    return &nws;
}

void NodeWrapper::Print() {
    cout << nws.myid << " " <<nws.pacrec << " " << nws.seqno << " "
<< nws.posx << " " << nws.posy << " "
        << (float) nws.txprob/RAND_MAX << " " << nws.rowid << endl;
}

void NodeWrapper::Initialize(int myid, int pacrec, int seqno, int posx,
int posy, int txprob, int* history, int rowid) {
    nws.myid = myid;
    nws.pacrec=pacrec;
    nws.seqno = seqno;
    nws.posx = posx;
    nws.posy = posy;
    nws.txprob = txprob;
    memcpy(nws.history, history, MAXNODES*sizeof(int));
    nws.rowid = rowid;
}

/*****Node Class*****/

```

```

Node::Node(int myid) {
    Initialize(myid, 0, 0, frand());
}

Node::Node(int myid, lfloat x, lfloat y, float txprob) {
    Initialize(myid, x, y, txprob);
}

Node::Node(nodewrapstruct* nws) {
    Initialize(nws->myid, (lfloat) nws->posx, (lfloat) nws->posy,
(float) nws->txprob/RAND_MAX);
    memcpy(history, nws->history, MAXNODES*sizeof(int));
    pacrec=nws->pacrec;
    seqno=nws->seqno;
}

Node::~Node() {
    delete history;
}

int Node::GetID() {
    return myid;
}

pos Node::GetPos() {
    pos pnew;
    pnew.x = p.x;
    pnew.y = p.y;
    return pnew;
}

float Node::GetTxProb() {
    return txprob;
}

void Node::Initialize(int myid, lfloat x, lfloat y, float txprob) {
    this->myid = myid;
    this->txprob = txprob;
    SetPos(x,y);
    seqno = 1;
    this->pacrec = 0;
    history = new int[MAXNODES];
        if (!history) {
            cout << "nullpointer returned from memory
reference" << endl;
            exit(1);
        }
}

int Node::GetPacketsReceived() {
    return pacrec;
}

void Node::SetPos(lfloat x, lfloat y) {
    p.x = x;

```

```

        p.y = y;
    }

Packet* Node::TxPacket(int dest, PACKETTYPE type) {
    if (frand() > txprob) {
        Packet* pack = new Packet(myid, (dest==myid) ? 0 : dest, 0,
type, seqno);
        if (!pack) {
            cout << "nullpointer returned from memory
reference" << endl;
            exit(1);
        }
#ifdef DEBUG
        cout << "node " << myid << " sent packet " << seqno << " to
" << (dest==myid) ? 0 : dest << endl;
#endif
        seqno++;
        return pack;
    }
    return NULL;
}

Packet* Node::RxPacket(Packet* pack) {
    if (history[pack->GetSource()]>=pack->GetSeqNo()) {
        if (pack->GetCurrTime() != zork2) {
            pack->SetCurrTime(zork2);
            pack->SetAlive(false);
            dellist.insert(pack);
        }
        return NULL;
    }
    else history[pack->GetSource()]=(history[pack->GetSource()] >
pack->GetSeqNo()) ?
        history[pack->GetSource()] : pack->GetSeqNo();
    /*
    if (pack->GetSource()==myid) {
        if (pack->GetCurrTime() != zork2) {
            pack->SetCurrTime(zork2);
            pack->SetAlive(false);
            dellist.insert(pack);
        }
        return NULL;
    }
    */
    if (pack->GetDest()!=myid&&(zork2-pack-
>GetStartTime())<MAXHOPCOUNT) {
        pack->SetHopCount(pack->GetHopCount()+1);
#ifdef DEBUG
        cout << "node " << myid << " received packet (" << pack-
>GetSeqNo() << ") from "
            << pack->GetSource() << " to " << pack->GetDest() <<
endl;
#endif
        pack->SetCurrTime(zork2);
        pack->SetAlive(true);
        return pack;
    }
}

```

```

        pacrec++;
#ifdef DEBUG
        cout <<"node "<<myid<<" RECEIVED packet ("<<pack-
>GetSeqNo()<<" from "
                <<pack->GetSource()<<" to "<<pack->GetDest() << endl;
#endif

        if (pack->GetCurrTime() != zork2) {
            pack->SetCurrTime(zork2);
            pack->SetAlive(false);
            dellist.insert(pack);
        }

        return NULL;
    }
}

int* Node::GetHistory() {
    return history;
}

int Node::GetSeqNo() {
    return seqno;
}

void Node::Print() {
    cout << "Node: myid " << myid << ", txprob " << txprob << ", pos
(" << p.x << ", " << p.y
        << "), seqno " << seqno << ", pacrec " << pacrec << endl;
}

bool Node::MoveRandom(int cellsize) {
    SetPos(p.x+1,p.y+1);
    if (p.x>=cellsize||p.y>=cellsize||p.x<0||p.y<0) return true;
    return false;
}

void Node::WrapPos(int cellsize) {
    if (p.x>=cellsize) p.x = p.x%cellsize;
    else if (p.x<0) p.x+= cellsize;

    if (p.y>=cellsize) p.y = p.y%cellsize;
    else if (p.y<0) p.y+= cellsize;
}

/*****Cell Class*****/
Cell::Cell(int size) {
    this->size = size;
}

Cell::~Cell() {
    if (GetCount() <=0) return;
    set<Node*>::iterator nit;
    for(nit = nodelist.begin();nit != nodelist.end();nit++)
        delete *nit;

    set<Packet*>::iterator pit;

```

```

        for(pit = outgoing.begin();pit != outgoing.end();pit++)
            delete *pit;

        for(pit = incoming.begin();pit != incoming.end();pit++)
            delete *pit;

        nodelist.clear();
        outgoing.clear();
    }

void Cell::AddNode(Node* ptr) {
    ptr->WrapPos(size);
    nodelist.insert(ptr);
}

void Cell::AddPacket(Packet* ptr) {
    if (GetCount() <=0) return;
    set<Node*>::iterator pit;
    for(pit = nodelist.begin();pit != nodelist.end();pit++)
        (*pit)->RxPacket(ptr);
}

Packet* Cell::GetPacket() {
    if (outgoing.size()==0) return NULL;
    Packet* p;
    set<Packet*>::iterator pit;
    pit=outgoing.begin();
    p = *pit;
    outgoing.erase(pit);
    return p;
}

int Cell::GetCount() {
    return (!nodelist.empty()) ? nodelist.size() : 0;
}

void Cell::MoveNodes(int row, set<NodeWrapper*>*
nleft,set<NodeWrapper*>* nright) {
    if (nodelist.empty()) return;
    set<Node*>::iterator nit;
    for(nit = nodelist.begin();nit != nodelist.end();nit++) {
        if ((*nit)->MoveRandom(size)) {
            if (this->GetBorder()) {
                int temp = 0;
                if ((*nit)->GetPos().y>=size) temp++;
                else if ((*nit)->GetPos().y<0) temp--;

                (*nit)->WrapPos(size);
                if (this->GetBorder()==RIGHT) nright-
>insert(new NodeWrapper(*nit, row+temp));
                else if (this->GetBorder()==LEFT) nleft-
>insert(new NodeWrapper(*nit, row+temp));
                nodelist.erase(nit);
            }
            else {
                Cell* c = this;
                if ((*nit)->GetPos().x>=size) c=c->right;
                else if ((*nit)->GetPos().x<0) c=c->left;
            }
        }
    }
}

```

```

        if ((*nit)->GetPos().y>=size) c=c-
>bottom;
        else if ((*nit)->GetPos().y<0) c=c-
>top;
        (*nit)->WrapPos(size);
        c->AddNode(*nit);
        nodelist.erase(nit);
    }
}

}

void Cell::Print(int i, int j) {
    cout << "Cell (" << i << ", " << j << "): count " << GetCount() <<
" " << (int) GetBorder() << endl;
    PrintList();
}

void Cell::Print() {
    cout << "Cell: count " << GetCount() << " " << (int)
GetBorder() << endl;
    PrintList();
}

void Cell::PrintList() {
    if (nodelist.empty()) return;
    set<Node*>::iterator pit;
    for(pit = nodelist.begin();pit != nodelist.end();pit++) {
        (*pit)->Print();
    }
}

void Cell::TxPackets(int dest, PACKETTYPE type) {
    if (GetCount() <=0) return;
    Packet* p;
    set<Node*>::iterator pit;
    for(pit = nodelist.begin();pit != nodelist.end();pit++) {
        p=(*pit)->TxPacket(dest,type);
        if (p) outgoing.insert(p);
    }
}

void Cell::RxPacket(Packet* p) {
    incoming.insert(p);
}

void Cell::RxAll() {
    if (GetCount() <=0||incoming.empty()) return;
    Packet* pnew;
    set<Node*>::iterator nit;
    set<Packet*>::iterator pit;
    for(nit = nodelist.begin();nit != nodelist.end();nit++) {
        for (pit = incoming.begin();pit!=incoming.end();pit++) {
            pnew=(*nit)->RxPacket(*pit);
            if (pnew) outgoing.insert(pnew);
        }
    }
}

```

```

    }
    incoming.clear();
}

void Cell::SetBorder(byte value) {
    isborder=value;
}

byte Cell::GetBorder() {
    return isborder;
}

int Cell::GetSize() {
    return size;
}

void Cell::GetData(int row, set<PacketWrapper*>* pw,set<NodeWrapper*>*
nw) {
    set<Node*>::iterator nit;
    for(nit = nodelist.begin();nit != nodelist.end();nit++) {
        nw->insert(new NodeWrapper(*nit, row));
        delete *nit;
    }

    nodelist.clear();

    set<Packet*>::iterator pit;
    for(pit = outgoing.begin();pit != outgoing.end();pit++) {
        pw->insert(new PacketWrapper(*pit, row,0,0,0));
        //delete *pit;
    }
    outgoing.clear();
}

/*****Framework Class*****/

Framework::Framework(int xcellssperproc, int meterspercell, int size,
int rank) {

    cells = new Cell*[dimension=(xcellssperproc)*(xcellssperproc)];
    if (!cells) {
        cout << "nullpointer returned from memory
reference" << endl;
        exit(1);
    }

    zork2=nodecount=0;
    timeit=timetotal=0;
    whichframe=rank;
    nprocs=size;
    for(int i = 0; i < dimension; i++) {
        cells[i] = new Cell(meterspercell);
        cells[i]->SetBorder(NONE);
        if (!cells[i]) {

```

```

        cout << "nullpointer returned from memory
reference" << endl;
        exit(1);
    }
}
for(int i = 0; i < dimension; i++) {
    if (i % (xcellsperproc)==0) {
        cells[i]->left = cells[i+(xcellsperproc-1)];
        if (nprocs>1) cells[i]->SetBorder(LEFT);
    }
    else cells[i]->left = cells[i-1];
    if (i % (xcellsperproc)==xcellsperproc-1) {
        cells[i]->right = cells[i-(xcellsperproc-1)];
        if (nprocs>1) cells[i]->SetBorder(RIGHT);
    }
    else cells[i]->right = cells[i+1];
    if (i < xcellsperproc) {
        cells[i]->top = cells[dimension-(xcellsperproc)+i];
    }
    else cells[i]->top = cells[i-xcellsperproc];
    if (i >= dimension-(xcellsperproc)) {
        cells[i]->bottom = cells[i-(dimension-
(xcellsperproc))];
    }
    else cells[i]->bottom = cells[i+xcellsperproc];
}
head = GetCell((lfloat) 0, (lfloat) 0);
dimension=xcellsperproc;
scale=meterspercell;
}

Framework::~Framework() {
    //if (cells!=NULL) delete cells;
}

void Framework::AddNode(Node* ptr) {
    Cell* c = GetGeoCell((ptr->GetPos()).x, (ptr->GetPos()).y);
    if (c) c->AddNode(ptr);
    nodecount++;
}

Cell* Framework::GetCell(lfloat x, lfloat y) {
    // cout << x << " " << y << endl;
    if (x < (lfloat) dimension && y < (lfloat) dimension) return
cells[x+y*dimension];
#ifdef DEBUG
    cout << "Getcell found invalid index" << endl;
#endif
    return NULL;
}

Cell* Framework::GetGeoCell(int x, int y) {
    if (x < dimension*scale && y < dimension*scale) return
GetCell((float)x/(dimension*scale), (float)y/(dimension*scale));
    return NULL;
}

```

```

Cell* Framework::GetCell(float x, float y) {
    return GetCell((unsigned int)floor((float) x*(dimension-1)+.5),
        (int unsigned)floor((float) y*(dimension-1)+.5));
}

void Framework::TxPackets(int dest, PACKETTYPE type) {
    lfloat i=0;
    bool row = true, col=true;
    for(Cell* crow=head;crow!=head||row;crow=crow-
>bottom,i++,row=false,col=true)
        for (Cell* ccol = crow; ccol!=crow||col; ccol=ccol-
>right,col=false)
            if (ccol) ccol->TxPackets(dest,type);
}

void Framework::RxPackets() {
    lfloat i=0;
    bool row = true, col=true;
    for(Cell* crow=head;crow!=head||row;crow=crow-
>bottom,i++,row=false,col=true) {
        for (Cell* ccol = crow; ccol!=crow||col; ccol=ccol-
>right,col=false) {
            if (ccol) ccol->RxAll();
        }
    }
}

void Framework::MoveNodesInternal() {
    int i=0;
    bool row = true, col=true;
    for(Cell* crow=head;crow!=head||row;crow=crow-
>bottom,i++,row=false,col=true)
        for (Cell* ccol = crow; ccol!=crow||col; ccol=ccol-
>right,col=false)
            if (ccol) ccol->MoveNodes(i,&nleft,&nright);
}

//yeah...i know...
void Framework::MoveNodesExternal() {
    set<NodeWrapper*> nlefttemp;
    set<NodeWrapper*> nrighttemp;
    set<NodeWrapper*>::iterator sit;

    if (whichframe%2==0) {
        MPISendNW(&nleft, (!whichframe) ? nprocs-1 :
whichframe-1);
        MPISendNW(&nright, (whichframe+1)%nprocs);

        for(sit=nleft.begin();sit != nleft.end(); sit++) delete
*sit;
        for(sit=nright.begin();sit != nright.end(); sit++)
delete *sit;

        MPIGetNW(&nrighttemp, (whichframe+1)%nprocs);
        MPIGetNW(&nlefttemp, (!whichframe) ? nprocs-1 :
whichframe-1);

```

```

        for(sit=nlefttemp.begin();sit != nlefttemp.end());
sit++) {
            Cell* c = head;
            for (int i = 0; i < (*sit)->GetNWS()->rowid;
i++) c=c->bottom;
                c->AddNode(new Node((*sit)->GetNWS()));
                delete *sit;
            }

        for(sit=nrighttemp.begin();sit != nrighttemp.end());
sit++) {
            Cell* c = head->left;
            for (int i = 0; i < (*sit)->GetNWS()->rowid;
i++) c=c->bottom;
                c->AddNode(new Node((*sit)->GetNWS()));
                delete *sit;
            }
        }
        else {
MPIGetNW(&nrighttemp, (whichframe+1)%nprocs);
MPIGetNW(&nlefttemp, (!whichframe) ? nprocs-1 :
whichframe-1);

        for(sit=nlefttemp.begin();sit != nlefttemp.end());
sit++) {
            Cell* c = head;
            for (int i = 0; i < (*sit)->GetNWS()->rowid;
i++) c=c->bottom;
                c->AddNode(new Node((*sit)->GetNWS()));
                delete *sit;
            }

        for(sit=nrighttemp.begin();sit != nrighttemp.end());
sit++) {
            Cell* c = head->left;
            for (int i = 0; i < (*sit)->GetNWS()->rowid;
i++) c=c->bottom;
                c->AddNode(new Node((*sit)->GetNWS()));
                delete *sit;
            }
        MPISendNW(&nleft, (!whichframe) ? nprocs-1 : whichframe-1);
        MPISendNW(&nright, (whichframe+1)%nprocs);

        for(sit=nleft.begin();sit != nleft.end(); sit++) delete
*sit;
        for(sit=nright.begin();sit != nright.end(); sit++)
delete *sit;
    }

    nodecount-=nleft.size()+nright.size();
    nodecount+=nlefttemp.size() + nrighttemp.size();

    nleft.clear();
    nright.clear();
}

bool Framework::DoesCrossFrames(Cell* c, int row) {

```

```

    Cell* templeft = c, *tempright=c;

    set<Packet*>::iterator pit;
    for (int i = 0; i < COMMRADIUS; i++,tempright=tempright-
>right,templeft=templeft->left) {
        if (tempright->GetBorder()|| templeft->GetBorder()) {
            if (tempright->GetBorder()==RIGHT) {
                for (pit=c->outgoing.begin();pit != c-
>outgoing.end(); pit++)
                    pright.insert(new PacketWrapper(*pit, row,
COMMRADIUS-i-1,COMMRADIUS,tempright->GetBorder()));
                break;
            }
            else if (templeft->GetBorder()== LEFT) {
                for (pit=c->outgoing.begin();pit != c-
>outgoing.end(); pit++)
                    pleft.insert(new PacketWrapper(*pit, row,
COMMRADIUS-i-1,COMMRADIUS,templeft->GetBorder()));
                break;
            }
        }
        else return false;
    }

/*
    set<PacketWrapper*>::iterator nit;
    cout << "pleft nodes" << endl;
    for (nit=pleft.begin();nit != pleft.end(); nit++)
        (*nit)->Print();

    cout << "pright nodes" << endl;
    for (nit=pright.begin();nit != pright.end(); nit++)
        (*nit)->Print();
*/
    return true;
}

void Framework::PropagateInternal(Cell* c) {
    Cell* temp = c, *temp2 = c;
    int i = 0, j = 0;

    for (i=0; i < COMMRADIUS; i++)
        temp = temp->top;

    for (i=0; i < COMMRADIUS&&temp->GetBorder() !=LEFT; i++) {
        temp=temp->left;
    }

    int horiz = i+1;

    for (i=0; i < COMMRADIUS&&temp2->GetBorder() !=RIGHT; i++) {
        temp2=temp2->right;
    }

    horiz += i;

    set<Packet*>::iterator pit;

```

```

        i=0;
        for (Cell* trow = temp; i < 2*COMMRADIUS+1; i++,trow=trow-
>bottom) {
            j=0;
            for (Cell* tcol = trow; j < horiz; j++,tcol=tcol->right) {
                if (tcol->GetCount() > 0)
                    for (pit=c->outgoing.begin();pit != c-
>outgoing.end(); pit++) {
                        tcol->RxPacket(*pit);
                    }
            }
        }
    }

void Framework::PropagateInternal(Packet* p, int startrow, int horiz,
int dir) {
    Cell* temp = head;
    int i = 0, j=0;
    for (i = 0; i < startrow; i++)
        temp = temp->bottom;

    for (i=0; i < COMMRADIUS; i++)
        temp = temp->top;

    if (dir==LEFT) temp=temp->left;

    i=0;
    for (Cell* trow = temp; i < 2*COMMRADIUS+1; i++,trow=trow-
>bottom) {
        j=0;
        for (Cell* tcol = trow; j < horiz+1; j++,(dir==RIGHT) ?
tcol=tcol->right : tcol=tcol->left)
            if (tcol->GetCount() > 0) tcol->RxPacket(p);
    }
}

void Framework::PropagateExternal() {
/*
    set<PacketWrapper*>::iterator nit;
    cout << "pleft nodes" << endl;
    for (nit=pleft.begin();nit != pleft.end(); nit++)
        (*nit)->Print();

    cout << "right nodes" << endl;
    for (nit=pright.begin();nit != pright.end(); nit++)
        (*nit)->Print();
*/
}

set<PacketWrapper*> temp;
if (whichframe%2==0) {
    MPISendPW(&pleft, (!whichframe) ? nprocs-1 : whichframe-1);
    MPISendPW(&pright, (whichframe+1)%nprocs);
    MPIGetPW(&temp, (whichframe+1)%nprocs);
    MPIGetPW(&temp, (!whichframe) ? nprocs-1 : whichframe-1);
}
else {
    MPIGetPW(&temp, (whichframe+1)%nprocs);
    MPIGetPW(&temp, (!whichframe) ? nprocs-1 : whichframe-1);
}

```

```

        MPISendPW(&pleft, (!whichframe) ? nprocs-1 : whichframe-1);
        MPISendPW(&pright, (whichframe+1)%nprocs);
    }

//    cout << whichframe << ": packets: " << pleft.size() << " " <<
pright.size() << endl;

    pleft.clear();
    pright.clear();

    set<PacketWrapper*>::iterator sit;
    for(sit=temp.begin();sit != temp.end(); sit++) {
        PropagateInternal(    new Packet((*sit)->GetPWS()),
        (*sit)->GetPWS()->rowid,
                                (*sit)->GetPWS()->over, (*sit)->GetPWS()-
>dir);
        delete *sit;
    }
}

void Framework::PropagateInternal() {
    lfloat i=0;
    bool row = true, col=true;
    for(Cell* crow=head;crow!=head||row;crow=crow-
>bottom,i++,row=false,col=true) {
        for (Cell* ccol = crow; ccol!=crow||col; ccol=ccol-
>right,col=false) {
            if (ccol) {
                if (!ccol->outgoing.empty()) {
                    if (nprocs>1)
DoesCrossFrames(ccol,i);
                PropagateInternal(ccol);
                ccol->outgoing.clear();
            }
        }
    }
}

void Framework::GarbageCollect() {
    set<Packet*>::iterator pit;
//    cout << "size is " << dellist.size() << " " << endl;
    for (pit = dellist.begin(); pit != dellist.end(); pit++) {
        if (!(*pit)->GetAlive()) {
            delete *pit;
            dellist.erase(pit);
        }
    }
}

void Framework::Iterate() {
    timeit = -MPI_Wtime();
    if (nodecount>0) TxPackets(random() % nodecount,PING);
    PropagateInternal();
    if (nprocs>1) PropagateExternal();
}

```

```

        RxPackets();
        if (nodecount>0) MoveNodesInternal();
        if (nprocs>1) MoveNodesExternal();
        if (nprocs>1) LoadBalance();
        GarbageCollect();
        timeit +=MPI_Wtime();
        timetotal += timeit;
        cout << whichframe << "," << zork2 << "," << nodecount<< "," <<
timeit << "," << timetotal << "," << dimension << endl;

        zork2++;
    }

void Framework::MPISendPW(set<PacketWrapper*>* myset, int dest) {
    set<PacketWrapper*>::iterator sit;
    packwrapstruct* array = new packwrapstruct[myset->size()];
    int sindex=0;
    for(sit=myset->begin();sit != myset->end(); sit++,sindex++)
        memcpy(&(array[sindex]), (*sit)->GetPWS(),
sizeof(packwrapstruct));
    sindex*=sizeof(packwrapstruct)/sizeof(int);
    // cout << "proc " << whichframe << " sending packet (" << sindex <<
") to " << dest << endl;
    // cout.flush();
    MPI_Send(&sindex, 1, MPI_INT, dest, MPIPACKETTYPE,
MPI_COMM_WORLD);
    // cout << "proc " << whichframe << " done sending packet (" <<
sindex << ") to " << dest << endl;
    // cout.flush();
    MPI_Send(array, sindex, MPI_INT, dest, MPIPACKETTYPE,
MPI_COMM_WORLD);
    delete array;
}

void Framework::MPIGetPW(set<PacketWrapper*>* myset, int source) {
    int rindex = 0;
    MPI_Status status;
    // cout << "proc " << whichframe << " rxing packet (" << rindex <<
") from " << source << endl;
    // cout.flush();
    MPI_Recv(&rindex, 1, MPI_INT, source, MPIPACKETTYPE,
MPI_COMM_WORLD, &status);
    // cout << "proc " << whichframe << " done rxing packet (" << rindex
<< ") from " << source << endl;
    // cout.flush();
    packwrapstruct* array = new
packwrapstruct[rindex/(sizeof(packwrapstruct)/sizeof(int))];
    MPI_Recv(array, rindex, MPI_INT, source, MPIPACKETTYPE,
MPI_COMM_WORLD, &status);
    for (int i = 0; i < rindex/(sizeof(packwrapstruct)/sizeof(int));
i++)
        myset->insert(new PacketWrapper(&array[i]));
    delete array;
}

void Framework::MPISendNW(set<NodeWrapper*>* myset, int dest) {
    set<NodeWrapper*>::iterator sit;

```

```

        nodewrapstruct* array = new nodewrapstruct[myset->size()];
        //cout << "myset size is " << myset->size() << endl;
        //cout.flush();
        int sindex=0;
        for(sit=myset->begin();sit != myset->end(); sit++,sindex++) {
            memcpy(&(array[sindex]), (*sit)->GetNWS(),
sizeof(nodewrapstruct));
        }

        sindex*=sizeof(nodewrapstruct)/sizeof(int);
        // cout << "proc " << whichframe << " sending npacket (" << sindex
<< ") to " << dest << endl;
        // cout.flush();
        MPI_Send(&sindex, 1, MPI_INT, dest, MPINODETYPE,
MPI_COMM_WORLD);
        if(COND) cout << "proc " << whichframe << " middle sending
npacket (" << sindex << ") to " << dest << endl;
        if(COND) cout.flush();
        MPI_Send(array, sindex, MPI_INT, dest, MPINODETYPE,
MPI_COMM_WORLD);
        if(COND) cout << "proc " << whichframe << " done sending npacket
(" << sindex << ") to " << dest << endl;
        if(COND) cout.flush();
        delete array;
    }

void Framework::MPIGetNW(set<NodeWrapper*>* myset, int source) {
    int rindex = 0;
    // cout << "proc " << whichframe << " rxing npacket (" << rindex <<
") from " << source << endl;
    // cout.flush();
    MPI_Status status;
    MPI_Recv(&rindex, 1, MPI_INT, source, MPINODETYPE,
MPI_COMM_WORLD, &status);
    if (COND) cout << "proc " << whichframe << " middle1 rxing
npacket (" << rindex << ") from " << source << endl;
    if (COND) cout.flush();
    nodewrapstruct* array = new
nodewrapstruct[rindex/(sizeof(nodewrapstruct)/sizeof(int))];
    if (COND) cout << "proc " << whichframe << " middle2 rxing
npacket (" << rindex << ") from " << source << endl;
    if (COND) cout.flush();
    MPI_Recv(array, rindex, MPI_INT, source, MPINODETYPE,
MPI_COMM_WORLD, &status);
    if (COND) cout << "proc " << whichframe << " done rxing npacket
(" << rindex << ") from " << source << endl;
    if (COND) cout.flush();
    for (int i = 0; i <
rindex/(sizeof(nodewrapstruct)/sizeof(int)); i++)
        myset->insert(new NodeWrapper(&array[i]));
    delete array;
}

int Framework::GetDimension() {
    return dimension;
}

```

```

void Framework::Print() {
    lfloat i=0,j=0;
    bool row = true, col=true;
    cout << "Framework: dimension " << dimension << endl;
    for(Cell* crow=head;crow!=head||row;crow=crow-
>bottom,i++,row=false,col=true,j=0) {
        for (Cell* ccol = crow; ccol!=crow||col; ccol=ccol-
>right,j++,col=false) {
            if (ccol) ccol->Print(i,j);
            else cout << "ccol (" << i << ", " << j << ") is nil"
<< endl;
        }
    }
}

int Framework::GetTimeStep() {
    return zork2;
}

void Framework::RemoveCol(set<PacketWrapper*>* pw, set<NodeWrapper*>*
nw, Cell* ptr) {
    if (ptr->left==ptr) return;
    dimension--;
    lfloat i=0;
    bool row = true;
    for(Cell* crow=ptr;crow!=ptr||row;i++,row=false) {
        if (head==crow) head=crow->right;
        if (crow->GetBorder()==LEFT) crow->right->SetBorder(LEFT);
        else if (crow->GetBorder()==RIGHT) crow->left-
>SetBorder(RIGHT);
        crow->GetData(i, pw, nw);
        crow->left->right = crow->right;
        crow->right->left = crow->left;
        Cell* temp = crow->bottom;
        delete crow;
        crow=temp;
    }
    nodecount-=nw->size();
}

void Framework::InsertCol(set<PacketWrapper*>* pw, set<NodeWrapper*>*
nw, Cell* ptr) {
    dimension++;
    lfloat i=0,j=0;
    bool row = true;
    for(Cell* crow=ptr;crow!=ptr||row;crow=crow->bottom,row=false)
    {
        Cell* newcell = new Cell(crow->GetSize());
        if (crow->GetBorder()==LEFT) {
            if (head==crow) head=newcell;
            newcell->left=crow->left;
            newcell->right=crow;
            crow->left = newcell;
            newcell->left->right=newcell;
        }
        else {
            newcell->left=crow;

```

```

        newcell->right=crow->right;
        crow->right = newcell;
        newcell->right->left=newcell;
    }
    newcell->SetBorder(crow->GetBorder());
    crow->SetBorder(NONE);
    newcell->bottom = NULL;
    newcell->top = NULL;
}
row = true;
i=0;
Cell* temp = (ptr->left->bottom==NULL) ? ptr->left : ptr->right;
for(Cell* crow=temp;crow!=temp||row;i++,row=false,crow=crow-
>bottom) {
    crow->bottom = crow->right->bottom->left;
    crow->top = crow->right->top->left;
}

nodecount+=nw->size();

set<NodeWrapper*>::iterator nit;
for(nit = nw->begin();nit != nw->end();nit++) {
    Cell* temp2 = temp;
    for (int i = 0; i < (*nit)->GetNWS()->rowid; i++)
        temp2=temp2->bottom;
    temp2->AddNode(new Node((*nit)->GetNWS()));
    delete *nit;
}

set<PacketWrapper*>::iterator pit;
for(pit = pw->begin();pit != pw->end();pit++) {
    Cell* temp2 = temp;
    for (int i = 0; i < (*pit)->GetPWS()->rowid; i++)
        temp2=temp2->bottom;
    temp2->outgoing.insert(new Packet((*pit)->GetPWS()));
    delete *pit;
}

pw->clear();
nw->clear();
}

void Framework::LoadBalance() {
    int nodesleft, nodesright, leftproc=(!whichframe) ? nprocs-1 :
whichframe-1, rightproc=(whichframe+1)%nprocs;
    int lbleftin=0,lbrightin=0,lbleftout=0,lbrightout=0;
    MPI_Status status;
    MPI_Send(&nodecount, 1, MPI_INT, leftproc, MPINODECOUNTTYPE,
MPI_COMM_WORLD);
    MPI_Recv(&nodesright, 1, MPI_INT, rightproc, MPINODECOUNTTYPE,
MPI_COMM_WORLD, &status);

    MPI_Send(&nodecount, 1, MPI_INT, rightproc, MPINODECOUNTTYPE,
MPI_COMM_WORLD);
    MPI_Recv(&nodesleft, 1, MPI_INT, leftproc, MPINODECOUNTTYPE,
MPI_COMM_WORLD, &status);
}

```

```

        if (nodesleft*1.20<nodecount&&dimension>=2) lbleftout=1;
        else if (nodesright*1.20<nodecount&&dimension>=2) lbrightout=1;

        MPI_Send(&lbleftout, 1, MPI_INT, leftproc, MPILBTYPE,
MPI_COMM_WORLD);
        MPI_Recv(&lbrightin, 1, MPI_INT, rightproc, MPILBTYPE,
MPI_COMM_WORLD, &status);

        MPI_Send(&lbrightout, 1, MPI_INT, rightproc, MPILBTYPE,
MPI_COMM_WORLD);
        MPI_Recv(&lbleftin, 1, MPI_INT, leftproc, MPILBTYPE,
MPI_COMM_WORLD, &status);

        if (lbleftout) {
            set<PacketWrapper*> pw;
            set<NodeWrapper*> nw;
            RemoveCol(&pw, &nw, head);
            MPISendNW(&nw, leftproc);
            MPISendPW(&pw, leftproc);
        }
        else if (lbleftin) {
            set<PacketWrapper*> pw;
            set<NodeWrapper*> nw;
            MPIGetNW(&nw, leftproc);
            MPIGetPW(&pw, leftproc);
            InsertCol(&pw, &nw, head);
        }

        if (lbrightout) {
            set<PacketWrapper*> pw;
            set<NodeWrapper*> nw;
            RemoveCol(&pw, &nw, head->left);
            MPISendNW(&nw, rightproc);
            MPISendPW(&pw, rightproc);
        }
        else if (lbrightin) {
            set<PacketWrapper*> pw;
            set<NodeWrapper*> nw;
            MPIGetNW(&nw, rightproc);
            MPIGetPW(&pw, rightproc);
            InsertCol(&pw, &nw, head->left);
        }

    }

/*****Packet Class*****/
}

```