Visualization of Parallel Programming:

A Tool for Understanding Message Passing in Parallel Systems

CSC 499, Senior Capstone Project

Advisor: Professor Valerie Barr

Andrew Schwartz

June 15, 2013

Abstract

As advances in computer architecture near atomic limitations, parallel programming using many computation nodes in tandem is quickly becoming the future of high performance computing. Recently, parallel programming has begun being taught in computer science classrooms all over the world. Understanding the complex interactions between computers while a parallel program executes is vital to maximizing performance of the distributed system. However, students tend to have difficulty comprehending parallel concepts because professors generally have to rely on static diagrams to explain these interactions. Static representation does not adequately express parallel system behavior as it is inherently dynamic. We developed a visualizer for a virtual cluster running MPI (Message Passing Interface) algorithms in order to aid student understanding of general parallel programming concepts. The visualizer aims to provide an intuitive interface for displaying the data flow of various MPI functions in a customizable cluster configuration while also capturing and displaying benchmarking data to show the costs and benefits of different configurations. Our intention is that students using our visualizer can quickly gain a firm grasp of parallel programming concepts, which will help them develop into the programmers of the future.

1 Report Summary

1.1 Foreword

The purpose of the Senior Capstone Project is to give the opportunity to work on a research project of the student's interests. Students are given two terms to research, prototype, develop, and analyze an application related to their topic of choice, culminating in a research paper. The type of project I decided on was an educational software application. In my case, I developed a keen interest in parallel programming after taking a course during my junior year. It was the first time this course was offered at Union, and the field is relatively young, so there are not as many educational resources available on this subject. As a result, I decided I would take on the task on developing an application that would help students who are just getting started learning about about parallel programming. The goal of this report is to summarize and showcase the development process of the application.

1.2 Summary

The problem was initially approached by discussing with the parallel programming professor about the type of application that would be beneficial for students. I began to map out what kind of information would be useful for students to see. When I decided on a visualizer for data flow, the problem became determining what kind of information from the data flow of a parallel program is beneficial for students. From this, I developed a mock up and continuously received feedback from both students and professors on the design. After creating a workable mock-up, I constructed the class hierarchy of the program based on what kind of information was necessary and the best possible way to encapsulate that data. After implementing a functional program, unfortunately there were no parallel programming students at the time, so I was unable to test my application wit the target audience. However, in the future, I hope to experiment with using the application to determine if the application was a success.

Contents

List of Figures

2 Introduction

2.1 Problem

Easy comprehension of the fundamental structure of a parallel computing network is a much sought after goal pursued by computer scientists, particularly those who wish to teach those unfamiliar with parallel systems. Attempted solutions include programs which log the execution path of a parallel system for review and visualizers which graphically display the interactions between nodes in a parallel system. Reasons why this type of program may be useful to scientists include but not limited to: profiling of program execution, academic understanding of parallel programming structure, and determining the correct network topology for a particular parallel algorithm to run most efficiently.

When first introduced to parallel programming, it can be difficult to comprehend how particular aspects of libraries such as MPI are implemented. With many different ways to perform the same calculation, the correct algorithm for a particular problem can be difficult to determine. This is in part due to the difficulty in visualizing the complex interactions of parallel functions within a distributed system which span across multiple nodes. Professors of parallel computing generally have to rely on static diagrams with arrows written on a chalk board to attempt to explain these interactions. However, data flow within a system cannot be expressed well as it is inherently dynamic, and flows from one node to another during execution. This problem has been recognized and attempts to create programs which visualize MPI functions have been made [3] [10] [8] [11], but each program is lacking in particular features.

2.2 Objective

The goal of the project was to combine both profiling data and visual cues to create a parallel network visualizer which can aid students in both understanding how data flows in a parallel network and deciding which network topology is best for a given programming problem. The focus was on creating an easy to use graphical interface that can animate the data flow of a virtualized parallel network and provide contextual information that can be used by students to gauge the relative efficiency of the network. Students should be able to build a custom network topology, and experiment with various MPI functions. Initially, this

aims to help the student understand the foundation of message passing within a network. However, through continuous experimentation, the student should be able to gain a firm understanding of the advantages and disadvantages of particular network topologies. The primary purpose of the tool is to aid in the learning and understanding of the processes that go on behind the scenes when using a parallel programming library and grant the knowledge to be able to make the best design decisions.

3 Background

3.1 Parallel Computing History

The traditional way of writing computer software has been serially, meaning that instruction execute one after another in order. Instructions are sent to the CPU one at a time, and after each instruction finishes execution, the next instruction may be executed. In the past, CPU performance has tightly followed an observation named Moore's law. Moore's law is the observation that the number of transistors on integrated circuits doubles approximately every two years. An extension of this says that the period for a doubling in chip performance is approximately 18 months, due to increased chip speed, and transistor count. However, as advances in computer architecture near atomic limitations, parallel programming is quickly becoming the future of computing. A single CPU can no longer keep up with the ever-increasing demands of computing. Furthermore, the power consumption of a processor is described by the equation $P = CxV^2xF$ where P is power, C is the capacitance change per cycle, V is voltage, and F is the frequency of the processor. Before parallel processing was popular, the traditional method of increasing computer performance was to increase clock rate and transistors. However, it is clear that eventually, the power consumption of the processor would outweigh the benefits of the increase in performance in commercial settings. Parallel computing uses multiple processors to simultaneously execute a program and solve a task by breaking down the program into parts that do not depend on one another to execute. The task is then split across the processors and executed in parallel. The speed-up achieved by using parallel programming is defined by Amdahl's Law, which states that the maximum speed-up of the program is equal to the reciprocal of the percentage of the program that can be parallelized. For example, in Figure 1, we can see that a 50% parallelizable program can only achieve

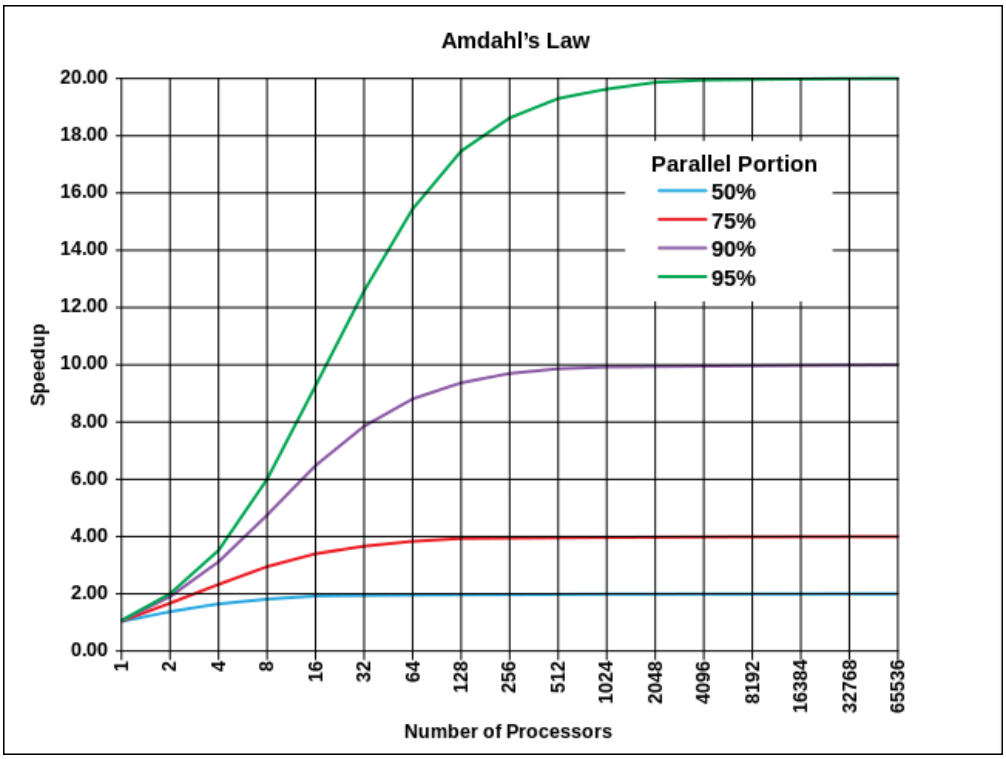

Figure 1: Amdahl's Law: Graph of the speed-up of a program with different levels of how much the program can be parallelized.

double the speed of a serial program, no matter how many processors there are. So, programmers are now rethinking how they create applications in order to maximize the percent parallelizable in their applications. Thus, an effort must be made to teach students the best parallel programming principles.

3.2 Message Passing and Distributed Computing

One of the ways of sending data between processors is through Message Passing. In the message passing paradigm, processes can send or receive messages to other processors. Messages can also be sent to synchronize processes. One of the benefits of using message passing as a mode of communication is that the processes do not have to be all running on the same machine, but rather can be mapped to a server address and accessed over the network. This allows for a parallel architecture design called distributed computing,

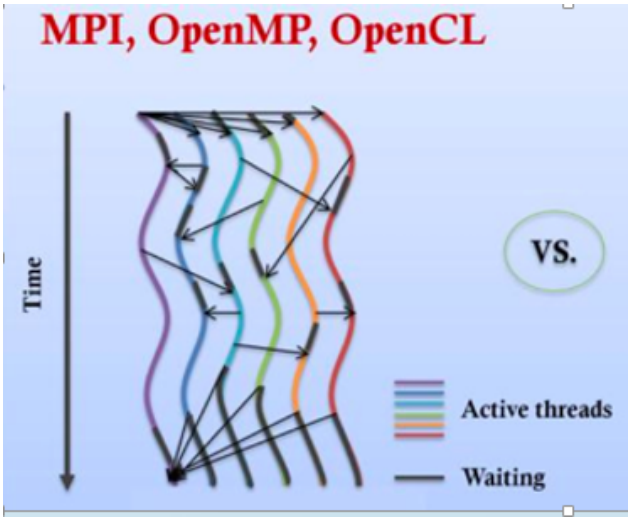

Figure 2: The figure above shows the flow of data in a typical program between different threads over time. Although in the example, threads are used, it applies directly to processes as well.

where many independent computers with their own set of hardware can synchronize and execute a parallel program. Distributed computing dramatically expands the possibilities of processor topologies for parallel programming. Instead of a single computer, any number of computers can now execute in parallel. Despite the radical change in network topology, we are still bound by Amdahl's law, so in order to maximize the potential of the new paradigm of computing, we need to completely understand parallel computational theory. However, visualizing the data flow of a network can be difficult, or even impossible without some visual aid. Even with visual aid (Figure 2), exactly how the program executes is not easy to grasp. So, to better understand message passing and the effects different network topologies have on performance, various applications have been developed.

3.3 MPIViz and ParaGraph

One of the applications previously developed for this purpose goes by the name of MPIViz (Figure 3). MPIViz fulfils its design goals well, but it leaves much to be desired in terms of aesthetics and functionality. The GUI, developed in Java Swing is simple, and runs well, but does not look very pleasing. At the same time, MPIViz offers very few options for determining efficiency. The options it does provide: customized

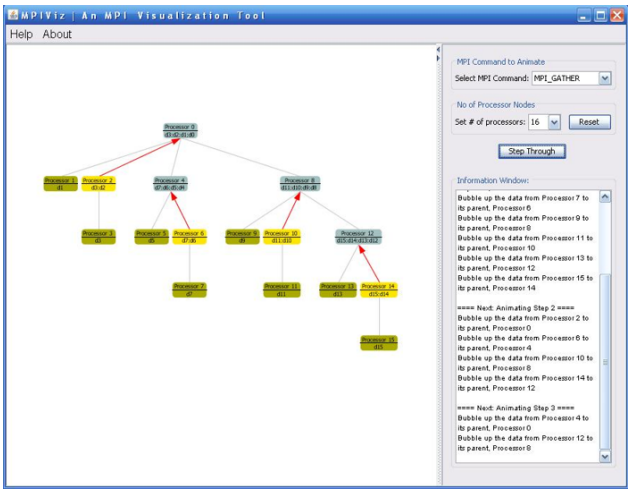

Figure 3: GUI screengrab of MPIViz, developed by Virginia Tech. The interface leaves much to be desired in terms of aesthetics and functionality.

data to be transferred and network configuration and great for visualization, but do not have the ability to determine if a particular network topology performs more efficiently than another, except for watching and estimating. On the other side of the spectrum, ParaGraph is another utility designed to profile MPI applications. Rather than being a visualizer, ParaGraph profiles real MPI programs by encapsulating and trapping all MPI calls. After execution, it outputs all of its profiling data in text windows (Figure **??**. ParaGraph does what MPIViz lacks, output profiling information. However, since it is not a visualization and requires an already written MPI program to run, it is not suited for educational purposes. ParaGraph was created for developers to gauge the efficiency of their already made programs, not to learn how data flows in a parallel network. The goal of this project is to build a visualization application similar to MPIViz, but extend it to provide relevant profiling information like those gathered by ParaGraph.

4 Design Requirements

In order to successfully complete the objective of the project, a number of specific design requirements must be met. The core design principles behind the project were:

Functionality The program must be able to accurately represent a user-customized network configuration.

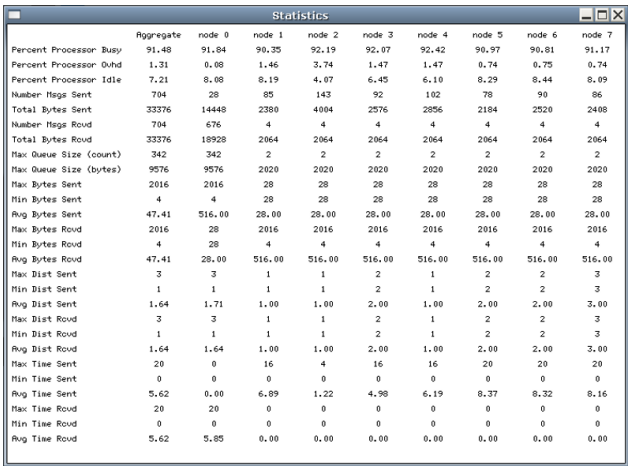

Figure 4: One of the many profiling output screens of ParaGraph. While it has a lot of good data, it is not too useful for educational purposes.

Furthermore, it must be able to emulate the data flow of a parallel program and present it in a step-by-step process to the user. Finally, the program must gather useful profiling statistics regarding the network configuration during execution and present it to the user.

Usability A user who is not familiar with parallel programming principles should be able to create a network and run an experiment with relative ease. The GUI should display information in an easy to understand manner and the user should be able to understand how to use the program effectively with limited reference to the user manual.

Extensibility The program should be able to extended to support more network configuration types and algorithms with limited code modification.

4.1 Functionality

4.1.1 Virtual Network

The virtual network will act as the model representation of the network within the program. The simulation and visualization with run on the virtual network. Contained within the virtual network model should be various parameters describing the network, including:

- Network Topology Type
- Number of Nodes
- Latency
- User Description

The parameters of the virtual network will determine the exact simulation of the network. The virtual network model will be acted on by events representing the various actions that are completed by the network when running a parallel program.

4.1.2 Events

Events are the key component to the simulation of the virtual network. When the program is asked to visualize a particular method, the program generates a list of events that represent the task in a step-by-step process. Types of events include:

Connection Event representing the connection request between two nodes on the virtual network.

Send Event representing the sending of data from one node to another.

Receive Event representing the receiving of data from one node to another.

When running a visualization, the events which represent the current simulation will be placed in an event manager to capture the vital information we need to display to the user.

4.1.3 Event Manager/Virtual Communications Layer

When a visualization begins, the event manager should be populated with the events that will be run on the network to simulate the algorithm properly. The event manager will also act as a virtual communications layer, trapping all events to store the profiling information for the user. Since events represent the data flow between nodes, the event manager is suitable for simulating the communication channels between the nodes. The event manager will read the set of parameters, such as latency, which can be tweaked by the user of the simulator to try and visualize communication over a true network of their choice. The event manager

will be the communication channel between nodes on the virtual cluster. By trapping the various events sent through the event manager, we can measure the overhead costs and other vital profiling statistics of the executing algorithm.

4.2 Usability

Because the objective of the program is to be used by those unfamiliar with parallel programming, it is important to keep a strong focus on the usability of the software. Software that is unable to be understood easily will fail in completing the goal of helping students more easily understand parallel programming principles.

4.2.1 I/O & GUI

Gathering input from the user should be a relatively simple process. All parameters for making a custom configuration should be readily available in an easy to use interface (Figure 10, 13, 14). Furthermore, immediate visual feedback on tweaked settings are important for maintaining a good user experience. A pleasant user experience while using the program is required for optimal learning. The goal is to get the user to try out the various settings of the application and a more complete understanding of the inner-workings of parallel programming. Just as important as the user input experience, the visualization must also be both informative and intuitive. The visualization should run with no lag or jitter, and be able to be tweaked by the user during execution. Also, the program should allow for quick analysis of the current performance of the network. Combining all of these ideas successfully should provide an intuitive interface suitable for a learning environment.

4.3 Extensibility

A final design requirement of the project was allowing for extensibility. From the beginning, there was a time constraint, so to allow for continued development, the application should be designed from the ground-up to be easy to extend. Knowing that all the features desired may not make it into the project under the time constraint, this can let the program evolve to meet other requirements as they come.

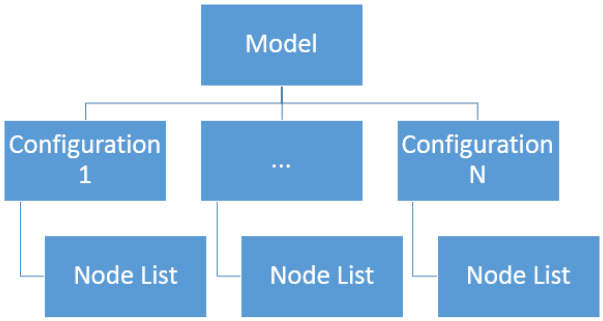

Figure 5: The class hierarchy of the classes used to represent a configuration. A model houses all the current configurations that the user has created. Each configuration then consists of a set of nodes.

4.3.1 Modular Design

The main method of allowing for extensibility was to write the code in modules. This separates the various parts of the program, allowing for easy modification and extension. Particularly, the goal of extensibility was to provide an easy way to add more visualization functionality without having to rewrite any code. Thus, the program will be written in an abstract manner that can be tailored to fit new design requirements as they are revealed.

5 Implementation

The project was written in C#.NET for its very good GUI builder and object oriented programming conducive to the design requirements of this project. In order to best meet the design requirements, a class was made to describe the various models and their containers. The main classes are an implementation of a Model, which houses Configurations, which consist of a set of Nodes. Each class should have a reference to its parent container and an ID. The class hierarchy can be seen below: Because the project can be classified into three classes: Model/View/Event, the implementation discussion will be structured as such.

5.1 Model

5.1.1 Model

The model is the parent class of an entire project. As users create configurations, they are placed into the model. Because the Model class is really just a container, its implementation is rather simple (Figure 6). Every configuration currently open will have a reference to the model of the current project. The Model implements the basic functionalities of a collection in order to add and remove Configurations. In the future, it may be possible to have more than one project open, in which case, a higher level of abstraction will be required to house the Models. However, for now, the user is restricted to one open project, with multiple configurations, at a time.

```
namespace vmpi
{
    public class Model
    {
        private List<Configuration> _configurations;
        public Model()
        {
            _configurations = new List<Configuration>();
        }

        public Configuration AddConfiguration(Configuration c)
        {
            _configurations.Add(c);
            return c;
        }

        public Configuration RemoveConfiguration(Configuration c)
        {
            _configurations.Remove(c);
            return c;
        }
```

Figure 6: This is an overview of the Model class. Notice how it has a single instance variable, a collection of Configurations. This precisely matches our literal definition of the Model class.

5.1.2 Configuration

The configuration is the container for Nodes. They are also the object representation of the visualization that is drawn on the screen. The overview for the Configuration object has been omitted from this section

because it is too large. It can be found in the Appendix as Figure 16 on page 29. The Configuration object also has instance variables for its defining characteristics. For instance, the latency of the configuration, or the type of configuration. These properties are important for calculating the profiling data and determining which algorithm to use to draw the Configuration.

5.1.3 Node

A node is the object representation of a single processor within parallel system. Nodes are contained within configurations. The overview for the Node object has been omitted from this section because it is too large. It can be found in the Appendix as Figure 17 on page 30. The Node object also has instance variables for keeping track of profiling and simulation data. It is important for the Node to keep track of which Node it is interacting with so the connection between them can be highlighted properly. In addition. each node has a property called NodeState. NodeState is an enumeration mapping the possible states of a node to integers so that the code is more readable when drawing. NodeState is shown in Figure 7.

```
public enum NodeState
{
    DISABLED = 0,
    COMPLETE = 1,
    ACTIVE = 2,
    WAITING = 3
}
```

Figure 7: Each node has a NodeState, which is used when painting the node to determine the proper color.

5.2 View

There are two integral aspects of the form: the customized ConfigurationTabPage and NodeLocator.

5.2.1 ConfigurationTabPage

The customized component ConfigurationTabPage is an extension of the general TabPage component. However, the ConfigurationTabPage houses a Configuration and on its paint method, paints the current configuration. It is in the paint method that the Configuration object is converted from a data model to a

visualization. The ConfigurationTabPage reads the properties of the Configuration and Nodes, and paints them accordingly. The ConfigurationTabPage becomes the canvas for our visualization, while maintaining the functionality of tabbed controls. This is beneficial because it allows for easy organization of open configurations.

5.2.2 NodeLocator

NodeLocator is a helper class that takes a Node an determines its coordinate location. This takes into account factors such as the number of nodes in the current configuration and the current size of the ConfigurationTabPage, because the ConfigurationTabPage can be resized. In this class, the 2D math is done to convert a node ID and Configuration to the proper screen coordinates for the Node. Trigonometry and bit shifting tricks are used to efficiently calculate the proper coordinates. The algorithm for the ring formation is shown below in Figure 8. The algorithm for trees is omitted from this section but can be found as Figure 18 in the Appendix on page 31.

```
case DrawStyle.RING:
    {

        int radius = Convert.ToInt32(((display.Height / 2) - (ringRect.Height / 2)) * 0.95);
        nodeXCoord = center.X - Convert.ToInt32(radius *
            Math.Cos(((n.id * (2 * Math.PI / ctp.parentConfiguration.GetNodeCount())))) + Math.PI / 2)) -
            (ringRect.Width / 2);
        nodeYCoord = center.Y - Convert.ToInt32(radius *
            Math.Sin(((n.id * (2 * Math.PI / ctp.parentConfiguration.GetNodeCount())))) + Math.PI / 2)) -
            (ringRect.Height / 2);
        break;
    }
```

Figure 8: The algorithm to determine where to draw a particular node in a ring formation.

5.3 Event

Events are the object that drive the visualization. Upon queueing a method, a list of events is generated corresponding to the current simulation. There are currently three types of events supported: SendEvents, RecvEvents, and StateEvents. Send and receive events are fairly straight forward and correspond to the sending and receiving of data. StateEvents are pseudo-events that occur separately from SendEvents and

RecvEvents that signal the change of a Node state. They are used by the visualization, but happen transparently and do not affect profiling information. An overview of the SendEvent class is shown below (Figure **??**. The SendEvent, as all events do, takes a variable amount of parameters. The first parameter is always the current configuration, and the next parameters come in pairs corresponding to the nodes interacting in that event. For example, a $SendEvent(c, 1, 2, 4, 7)$ corresponds to a configuration c, while node 1 sends data to node 2 and node 4 sends data to node 7. Every event has a method called *run*, which is called when the event is fired. When an event is fired, profiling and visualization information is updated in the model and the view is refreshed. The SendEvent *run* method is available in the Appendix on page 32.

```
public SendEvent(Configuration c, params object[] args)
{
    fromTo = new List<Tuple<int, int>>();
    for (int i = 0; i < args.Length; i+=2)
    {
        fromTo.Add(new Tuple<int, int>((int)args[i], (int)args[i + 1]));
    }
    this.c = c;
}
```

Figure 9: The SendEvent (and all events) take a variable amount of parameters. The first parameter is always the parent configuration, and pairs of parameters follow referencing which nodes are performing the action. The interacting nodes are stored in an instance variable titled fromTo.

The GUI was built using Microsoft Visual Studio. Designing and implementing these components together in this fashion allows for high extensibility and easy to read code. The final design succeeded in matching the design from the mock-up trials.

6 Performance as an Educational Tool

Unfortunately, due to the parallel programming course not being available for the past two terms, the target consumer of the application was unable to be found. As a result, a measurement of how well it performs as a teaching aid was impossible to perform. Volunteers from all disciplines were invited to test the program and informally gauge its usability and functionality. Although preliminary tests with students are promising, a full-fledged study will have to be left for the future when proper candidates are available.

7 Schedule

7.1 Brainstorming

The problem was initially approached by discussing with the parallel programming professor about the type of application that would be beneficial for students of the class. When a visualizer was decided as the project type, the problem became determining what kind of information from the data flow would be beneficial. The consultations spanned over a few days and from them, the kind of visualizer was decided on.

7.2 Mockup

Next, a mock up was developed and continuously received feedback from both students and professors on the design. The iterative process allowed for the creation of the most usable design. The mockup was developed in Balsamiq, so it was both easy to create and edit.

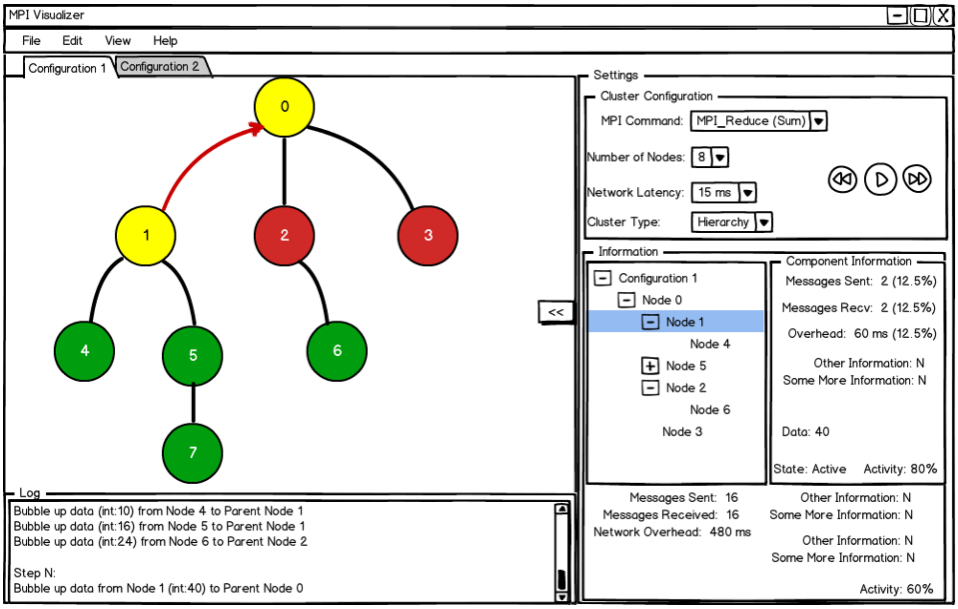

Figure 10: Application Mock-Up designed in Balsamiq.

7.3 Implementation

After creating a workable mock-up, I constructed the class hierarchy of the program based on what kind of information was necessary and the best possible way to encapsulate that data. After working out the class diagrams and program structure, there was a six week implementation phase where the functionality was developed. Afterwards, there was a small testing phase before a project demonstration. The implementation phase is outlined below in Figure **??**. In hindsight, development could have been started sooner. As a result

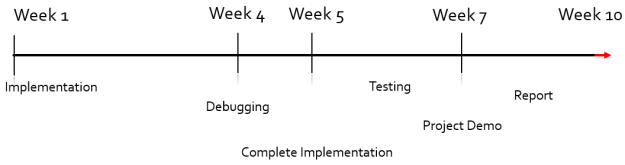

Figure 11: The schedule of tasks for the second term.

of a late development start, some quality of life functionalities are left out of the program. Because of this, the design goal of usability and functionality were less successful.

8 User Manual

Please refer to Figure 12 on page 25 when using the user manual.

1. Configuration Display: This is where the configuration will display. Using tabs at the top, you may change the active configuration. This is also visualization the simulation will take place.

2. Log: This will log all changes you make to the configuration. It is resizeable, so you may show as little or as much logging as you desire. The Configuration Display will enlarge to fill the space.

3. Node List: This will give a tree listing of the nodes in the current configuration. Selecting a node here will populate the Node Metrics (12).

4. Event Queue Box: This box will show the events left to be run on the current configuration. It is not necessary, and in a future build will probably be taken out.

5. Overall Metrics: The information here is the profiling data for the entire configuration over the current visualization.

6. Cluster Configuration Settings: Here you may design your customized cluster using the labeled commands.

7. Update Button: After customizing your configuration, press this button or the enter key to update your configuration.

8. Method Select Box: Here you may choose which MPI method to visualize.

9. Queue Button: After selecting the method, press this button to queue it up and populate the event queue.

10. Run Method: While this checkbox is checked, the simulation will fire events from the Event Queue Box. Uncheck it to pause visualization,

11. Visualization Speed Slider: Adjust how quickly the events are fired from the Event Queue Box.

12. Node Metrics: This information is populated when selecting a node from the Node List (4), during or after a visualization.

In addition to these, there are context menus on the configuration tab and a menu bar with standard commands. However, the menu bar is currently not functional, this quality of life functionality will be implemented in a later version.

9 Conclusion

Easy comprehension of the fundamental structure of a parallel computing network is a much sought after goal pursued by computer scientists, particularly those who wish to teach those unfamiliar with parallel systems. Attempted solutions include programs which log the execution path of a parallel system for review and visualizers which graphically display the interactions between nodes in a parallel system. Reasons why this type of program may be useful to scientists include but not limited to: profiling of program execution,

academic understanding of parallel programming structure, and determining the correct network topology for a particular parallel algorithm to run most efficiently. In response, an application was developed to attempt to solve this problem. The goal was to research and develop an educational application to aid students in easily understanding data flow through a network under different conditions. I believe that the program is successful in visualizing data flow and with a few more tweaks, will be capable of helping students studying parallel programming.

9.1 Improvements and Future Work

In the future, it will be necessary to determine if the visualization technique presented here is beneficial to students learning parallel programming. Volunteers from all disciplines were tested the program and informally gauged its usability and functionality. Although preliminary tests with students are promising, a full-fledged study will have to be left for the future when proper candidates are available. Further improvements on the design include finishing the quality of life functionalities such as saving configurations, and removing unnecessary UI components. Also, because of the heavy focus on modular design, further improvements can be made by incorporating more algorithms and methods from MPI into the visualizer.

References

[1] Patrick R. Amestoy, Abdou Guermouche, and Stephane Pralet Jean-Yves LExcellent. Hybrid scheduling for the parallel solution of linear sys- tems. *Parallel Comput.*, 2006.

[2] Rajive Bagrodia, Ewa Deelman, Steven Docy, and Thomas Phan. Performance prediction of large parallel applications using parallel simulations.

[3] Bedare and Tripathi. Mpiviz an mpi visualization tool, May 2008. http://research.cs.vt.edu/AVresearch/MPI/.

[4] Walfredo Cirne and Francine Berman. A model for moldable supercomputer jobs. In *Proceedings of the 15th International Parallel and Distributed Pro- cessing Symposium.*

[5] R.G. Covington, S. Dwarkada, J. R. Jump, J. B. Sinclair, and S. Madala. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1991.

[6] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, and V. Volkov. Parallel computing experiences with cuda. In *Micro, IEEE.*

[7] Stefan Harfst, Alessia Gualandris, David Merritt, Rainer Spurzem, Simon Portegies Zwart, and Peter Berczik. Performance analysis of direct n-body algorithms on special-purpose supercomputers. *New Astronomy*, 2007.

[8] Michael T. Heath. Recent developments and case studies in performance visualization using paragraph. In *Proceedings of the workshop on performance measurement and visualization on Performance measurement and visualization of parallel systems*, pages 175–200, Amsterdam, The Netherlands, The Netherlands, 1993. Elsevier Science Publishers B. V.

[9] Michael T. Heath. Performance visualization with paragraph. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 221–230, 1994.

[10] Michael T. Heath and Jennifer A . Etheridge. Visualizing the performance of parallel programs. *Software, IEEE*, 8(5):29–39, 1991.

[11] M.T. Heath. Visual animation of parallel algorithms for matrix computations. In *Distributed Memory Computing Conference, 1990., Proceedings of the Fifth*, volume 2, pages 1213 –1222, apr 1990.

[12] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, and N.J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom)*.

[13] Laxmikant V. Kale, Sameer Kumar, and Jayant DeSouza. A malleable job system for timeshared parallel machines. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*.

[14] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in multiprogrammed parallel systems. *SIGMETRICS Perform. Eval. Rev.*, 1988.

[15] Cathy McCann and John Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*.

[16] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba. Parallel processing of matrix multiplication in a cpu and gpu heterogeneous environment. *Lecture Notes in Computer Science*, 2007.

[17] Basile Schaeli, Sebastian Gerlach, and Roger D. Hersch. A simulator for parallel applications with dynamically varying compute node allocation. In *Proceedings of the 20th international conference on Parallel and distributed processing,*.

[18] Gladys Utrera, Julita Corbalan, and Jesus Labarta. Implementing malleability on mpi jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*.

[19] Nan Zhang, Yun shan Chen, and Jian li Wang. Image parallel processing based on gpu. In *2nd International Conference on Advanced Computer Control (ICACC)*.

10 Appendix

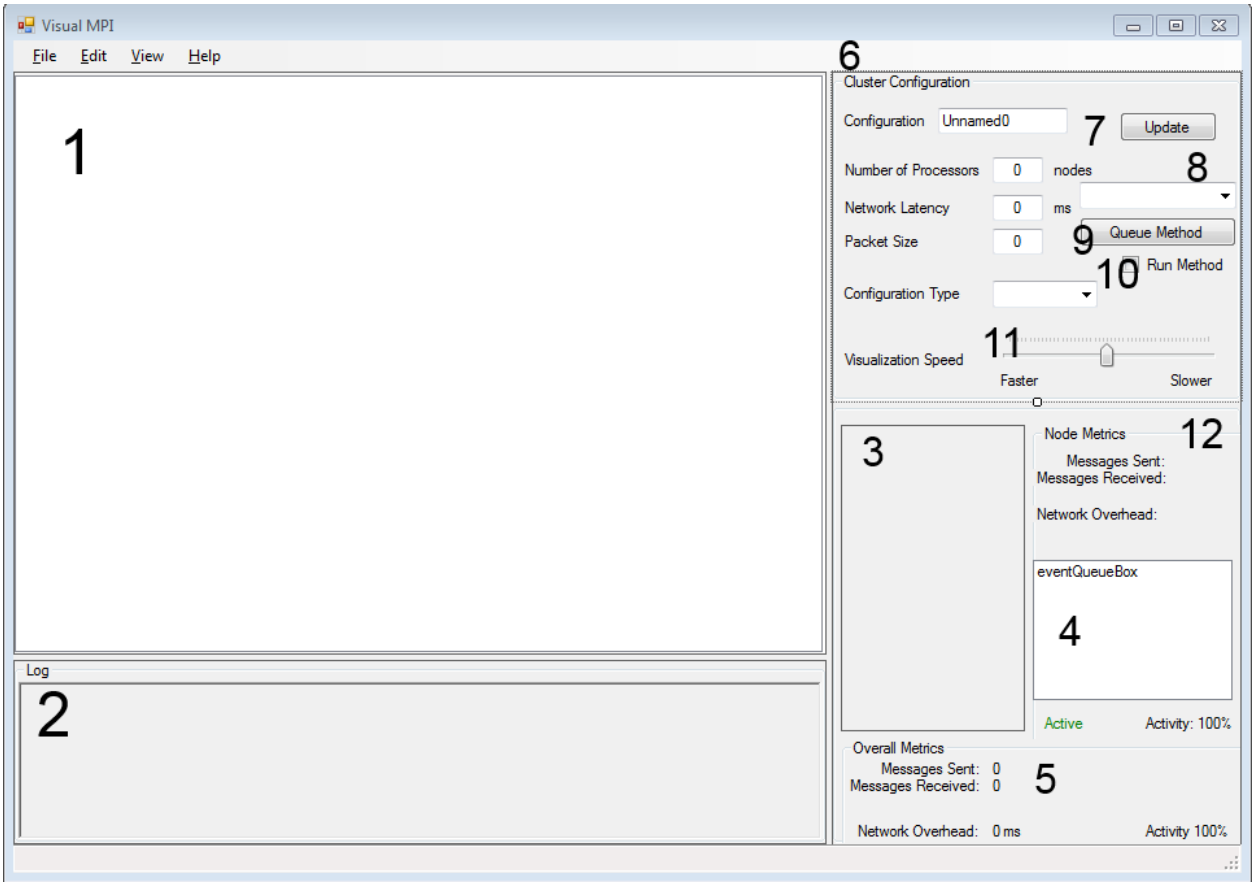

Figure 12: Reference figure for the user manual. Please refer to the numbers on this figure when using the user manual.

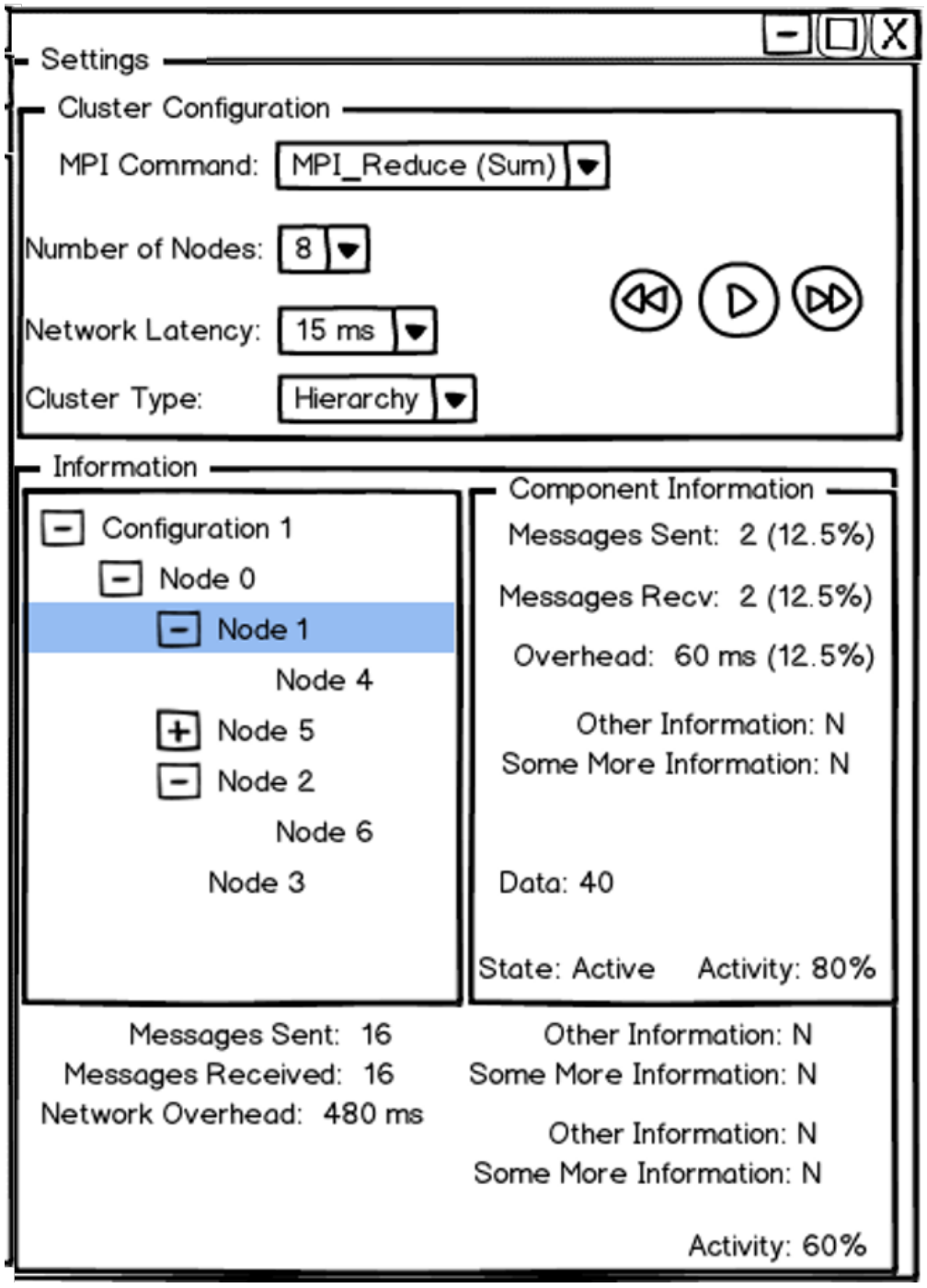

Figure 13: Application Mock-Up showcasing the proposed information panel. This panel would show the important profiling information to the user.

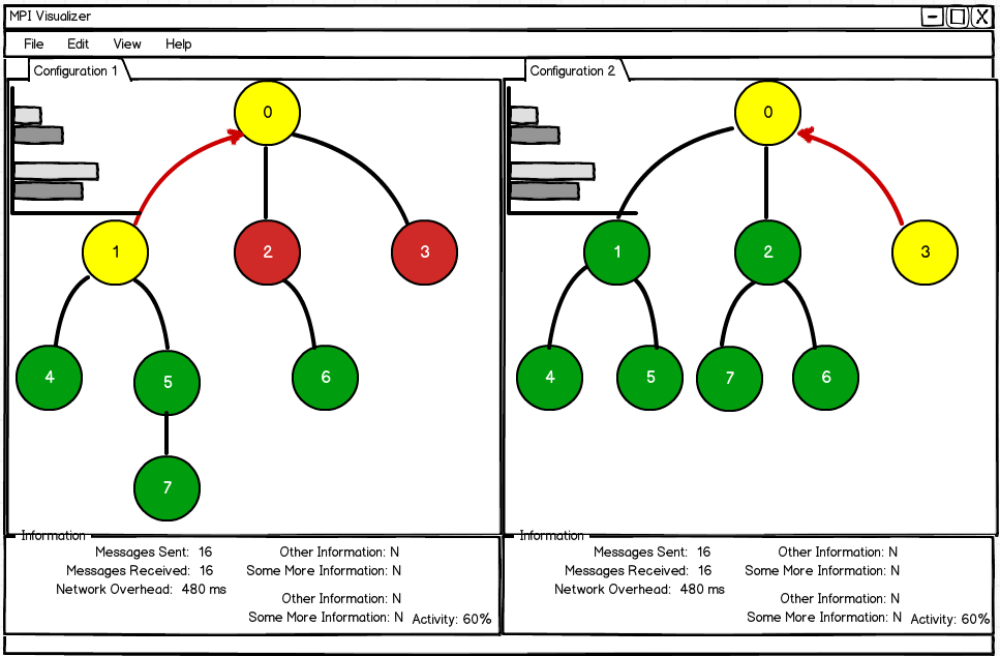

Figure 14: Application Mock-Up showing a proposed feature: side by side viewing. The visualization would run on both configurations simultaneously.

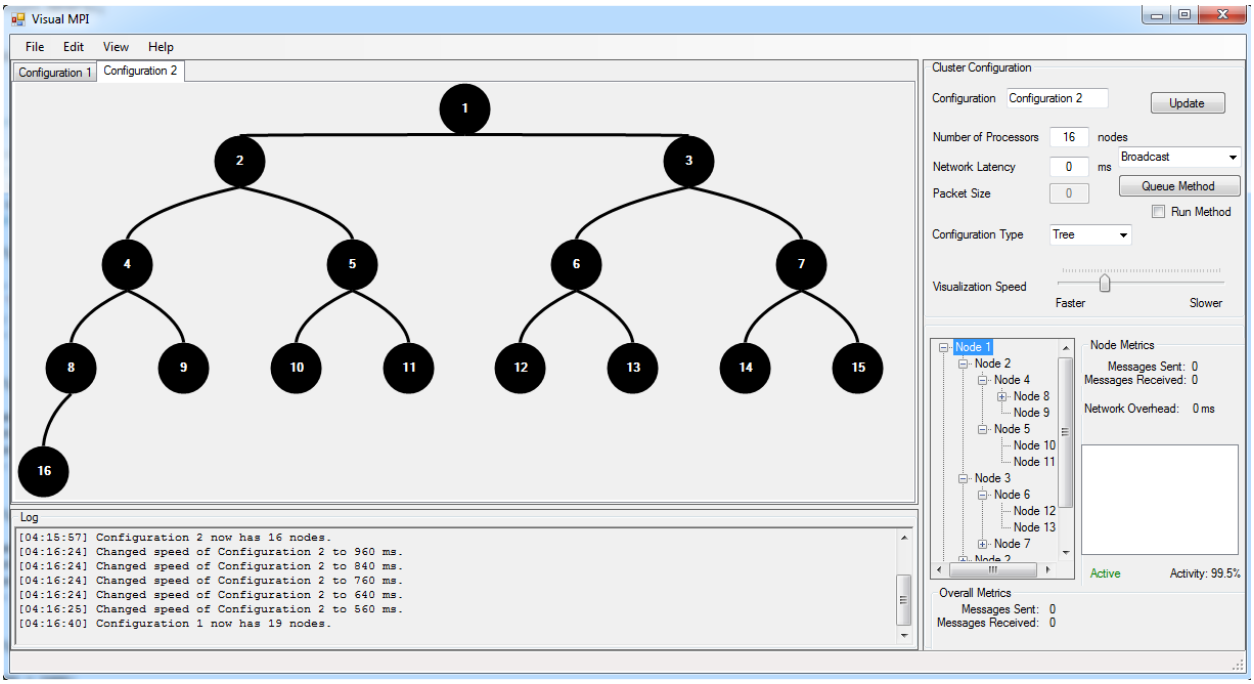

Figure 15: The application can be resized and adjusted based on user input andn the visualization will adjust. This should lead to a more pleasant user-customized experience.

```
namespace vmpi
{
    public class Configuration
    {
        public List<Node> nodes { get; private set; }
        public String name { get; set; }
        public int latency { get; set; }
        public int speed { get; set; }
        public DrawStyle configType { get; set; }

        public Configuration()
        {
            nodes = new List<Node>();
            name = "Unnamed0";
            latency = 0;
            speed = 25 * 40;
        }

        public Configuration(String name)
        {
            this.name = name;
            this.latency = 0;
            this.speed = 25 * 40;
            nodes = new List<Node>();
        }
```

Figure 16: This is an overview of the Configuration class. Similar to the Model class, it houses a collection of nodes. In addition, the user-specified parameters describing the Configuration are also instance variables.

```
namespace vmpi
{
    public class Node
    {
        private Configuration _parent;

        public int id { get; private set; }

        public NodeState state { get; set; }

        public int sentMsg { get; set; }

        public int recvMsg { get; set; }

        public double activity { get; set; }

        public Point location { get; set; }

        public int activeWith { get; set; }

        /// <summary>
        /// Creates a node and adds it to a parent Configuration.
        /// </summary>
        /// <param name="parent">Configuration to hold the node.</param
        /// <param name="id">ID for the node, usually the current node
        public Node(Configuration parent, int id)
        {
            this._parent = parent;
            this.id = id;
            this.state = NodeState.DISABLED;
            this.sentMsg = 0;
            this.recvMsg = 0;
            this.activity = 0.0;
            this.activeWith = -1;
        }
```

Figure 17: This is an overview of the Node class. Each node tracks its own profiling data. It is updated when the event is fired. In addition, each node has an ID and a reference to its parent configuration.

```
case DrawStyle.TREE:
    {
        int totalNodes = n.getParentConfiguration().GetNodeCount();
        int totalRows = 0;

        int row = 0;
        int col = 0;

        if (totalNodes == 1 || n.id == 1)
        {
            col = 0;
            row = 1;
            totalRows = 1;
        }
        else
        {
            while ((totalNodes >>= 1) >= 1)
            {
                totalRows++;
            }

            int i = 1;

            while (i <= n.id)
            {
                row++;
                i <<= 1;
            }

            col = n.id - (i >> 1);
        }
        int rowHeight = Convert.ToInt32(display.Height / totalRows);
        nodeXCoord = (1 + (col << 1)) * display.Width / (1 << row) - (ringRect.Width >> 1);
        nodeYCoord = rowHeight * (row - 1) - ((n.id == 1) ? 0 : (ringRect.Height));
        break;
    }
```

Figure 18: The algorithm to determine where to draw a particular node in a tree formation.

```
public Configuration run()
{
    foreach (Tuple<int, int> pair in fromTo)
    {
        Node from = c.nodes[pair.Item1-1];
        Node to = c.nodes[pair.Item2-1];
        from.sentMsg++;
        to.recvMsg++;
        from.state = NodeState.ACTIVE;
        to.state = NodeState.ACTIVE;
        from.activeWith = to.id;
        to.activeWith = from.id;
    }
    return c;
}
```

Figure 19: The run method of a SendEvent; this is what is called when the event is fired. The method goes through the instance variable populated earlier with interacting nodes and performs the operation, updating the profiling and visual information.