5-2010

A Visualization approach for message passing in parallel computing systems

Arunkumar Sadasivan
University of Nevada Las Vegas

A VISUALIZATION APPROACH FOR MESSAGE PASSING IN PARALLEL COMPUTING SYSTEMS

by

Arunkumar Sadasivan

Bachelor of Engineering
Anna University, Chennai India
2005

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2010

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Arunkumar Sadasivan

entitled

A Visualization Approach for Message Passing in Parallel Computing Systems

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
School of Computer Science

Jan B. Pedersen, Committee Chair

Kazeem Taghva, Committee Member

Yoohwan Kim, Committee Member

Henry Selvaraj, Graduate Faculty Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

May 2010

ABSTRACT

A Visualization Approach for Message Passing in Parallel Computing System

by

Arunkumar Sadasivan

Dr. Jan B. Pedersen, Examination Committee Chair
Assistant Professor, Department of Computer Science
University of Nevada, Las Vegas

Information Visualization has been used as an effective method in transforming textual information into a visual form that enables users to effectively communicate and understand the information. MPI (Message Passing Interface) usually involves a large amount of data, which necessitates exploring innovative ideas to visualize such data effectively.

In this thesis, we implement a graph visualization tool called MPROV to effectively visualize communication patterns of message passing using log files from MPI applications. The tool provides several interaction techniques to effectively reduce the amount of data displayed on the screen. We have also developed protocol conformance checking, which verifies the correctness of the communication by using a protocol specification file containing a list of constraints. Finally, we test our application with different log files containing a list of communication patterns and also at different strictness level that the messages should conform to based on the protocol constraints.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Advancement in computer technology led to the growth of human computer interaction (HCI), a research that focuses on interaction between human and computational machines to enable the viewer to observe and understand information. Information visualization is a sub-discipline that emerged due to research in human computer interaction and computer graphics; it has been used as an efficient approach to solve many real world problems involving huge amount of data. Information visualization opened a way to present large amounts of data in a manner that assist the user better in finding patterns and trends as well as analyzing the data and discovering unknown relationships. As any other visualization systems, it also uses visual cues to explore, analyze and communicate ideas in an effective manner.

In this thesis, we present a graph visualization tool that displays space-time diagrams representing the communication taking place in a message passing system. The messages are denoted by a directed acyclic graph with nodes representing the events (sending or receiving a message) and the edges representing the relations (program flow or message flow). The application execution is partitioned into sub graphs called epochs, containing a sequence of consecutive events in the program trace. The tool enables the user to filter the processes to be displayed on the screen thus providing a detailed view of the point (process) of interest; also it contains a zoom feature to adapt the graph to any level of detail as needed by the user. Furthermore, we have implemented a tool for checking messages against a communication protocol specification. This lead to a technique called ‘Protocol Conformance Checking’. The idea behind this technique is to have the user

write a specification file containing a number of constraints and using this information to check the correctness of the message passing. A protocol specification can be checked using different levels of strictness from level 0 being the basic, or default level to level 3 being the most restrictive requiring each message to satisfy exactly one constraint.

Graph Visualization – Message Passing

The graph visualization tool MPROV described above is mainly used to visualize message passing log data from MPI [2] applications. The log files are obtained during a data collection phase (while the program is running) that logs all communication events in the application. It can be used to construct the communication graph. The graphs in question contain epochs, which we can obtain by running the buffer allocation approximation algorithm from [3]. An MPI application contains message buffers to hold the contents of the messages to be delivered. If there are many messages being sent, then all available buffers may be exhausted leading to a deadlock. Hence, determining the minimum amount of buffers needed is necessary; this is formally defined by Brodsky et al. as the Buffer Allocation Problem (BAP) [3]. Solving BAP is NP-Complete, but a novel approach using epochs for an approximation has been described in [4].

The tool uses graph visualization techniques to represent the structural information as diagrams of graphs. These graphs are similar to Lamport space-time diagrams [5], which represent a casual ordering of messages in a message passing system. As an MPI application can involve a large number of processes, the graph layout may be dense. Though it is technically possible to display all the processes, non-availability of large screen size makes it cluttered if displayed on a standard display device. One potential

solution to the above problem is to use some powerful interaction techniques to navigate through the graphs. MPROV contains interaction techniques like range filter, pickup filter and showall filter to restrict the number of process components displayed on the screen.

Organization of this Thesis

An overview of graph visualization, protocol conformance checking technique, the Buffer Allocation Problem and different visualization tools are given in Chapter 2. In Chapter 3, we discuss in detail the implementation of the graph visualization tool and its usage. Chapter 4 describes the results with different log data and protocol specification files. It also includes several snapshots of the tool which illustrates the look and feel of the software. Finally Chapter 5 presents conclusions and recommendations for future work.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

In this chapter, we present a brief review of how visualization techniques could be used for problem solving related to huge datasets. Firstly, we will be discussing about graph visualization techniques related to MPI applications. Secondly, we discuss an epoch based approach that runs the approximation algorithm to display the communication patterns containing epochs. The BAP (Buffer Allocation Problem) is the problem of determining the minimum number of buffers to ensure deadlock free execution. Finally we present a protocol conformance checking technique that assists the user in checking that a message satisfies constraints specified in a protocol specification.

Graph Visualization

One of the key issues in graph visualization that needs to be addressed is the size of the graph that needs to be visualized. Large graphs not only lead to performance issues but also usability or viewability issues even if we could layout and display all the elements. It is a well known fact as defined in [6] that it is easy to perform a detailed analysis of data in graph structures when the size of the graph is relatively small. As the size increases, overall structure becomes complex making it difficult to comprehend.

Most techniques available are applicable only for relatively small graphs and are less relevant for larger graphs. The size of the graph can make a normal good layout algorithm unusable as it does not guarantee proper scaling when a large number of nodes are to be displayed. The layout may become very dense thus making interaction with the graph difficult. The first step in visualization is to find an effective method to reduce the

size of the graph to be displayed. Another important issue is time complexity. The visualization system needs to update information in real-time, within a very short period to time to ensure that the user does not notice the delay.

In order to reduce the visual complexity of a graph, it is necessary to reduce the number of visible elements displayed on the screen. Limiting the number of visual elements not only improves the clarity but also increases performance layout and rendering [8]. Various “abstraction” and “reduction” techniques [8] have been applied by researchers in order to reduce the visual complexity of a graph. They are usually done by discarding trivial or uninteresting information while preserving important features and structures in the graph. Approaches such as clustering and filtering may be used to reduce graph complexities. Clustering is the process of grouping a set of physical or abstract objects into classes of similar objects based on some chosen semantics. Filtering refers to removal of elements from the view, while hiding is a process of simply not displaying the un–selected nodes, also referred to as folding [8].

Visualization and Analysis of MPI applications

Graph based visualization tools or systems have been proposed to assist parallel computing since visual structures and relationships are easy to comprehend. Most of these systems are graph based, as multidimensional directed graph closely resembles the execution of parallel programs. We provide a brief review of three graph based visualization tools namely VAMPIR [12], VISPER [10] and PVMbuilder [21]. VAMPIR was mainly developed for performance analysis of parallel programs while VISPER is used for developing the communication graph in real time and PVMbuilder is used for

developing message passing programs. Though these tools are not closely related to our tool, they provide a few features such as filtering, interactive zoom for detail view, color coding to differentiate the different activities which are also provided in our tool.

The visualization environment VAMPIR is based on the research tool PARvis [17]. VAMPIR translates a given trace file into a variety of graphical views such as state diagrams, activity charts, time-line displays, and statistics to provide a reasonable basis or system understanding and program optimization. It contains an animation mode to locate bottlenecks and flexible filter operations to reduce the amount of information displayed along with a powerful zooming feature to identify problems at any level of detail.

VISPER is a software visualization tool for developing parallel programs which relies on the Message Passing Interface (MPI) [2] and Parallel Virtual Machine (PVM) [18]. The tool is graph-based and correlates both the control and data flow graphs into a Process Communication Graph (PCG), without a need for complex textual annotation. In Figure 2-1, an arc in a PCG denotes the data flow between the nodes. In this example, Process P_0 sends data to process P_1 which in turn sends data to process P_2 and finally process P_0 receives data from P_2.

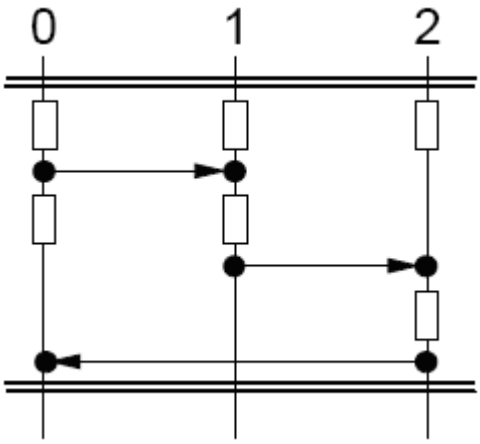

Figure 2-1: Program Structure and PCG.

The advantage of this approach is that no special language or compiler has to be used when programming. The tool simplifies writing and helps the understanding of parallel programs by allowing the programmer to explicitly specify communication, synchronization, and parallelism by drawing a Process Communication Graph. The fundamental idea behind VISPER is that programmers can create message-passing programs by drawing and annotating graphs.

PVMbuilder was created to construct message passing programs, as errors in the program might lead to unpredictable results such as deadlock, process failures and incorrect results. One of the main advantages of PVMbuilder is that it is not only used for program creation but also for debugging and program evolution. PVMbuilder can also be used for performance monitoring and tracing. It provides a high level graph structure which is the abstract representation of the program. The nodes in the graph are used to represent processes and the arcs denote the communication or control flow. The graph described in our tool is similar to the graph structure mentioned above.

Buffer Allocation in Message Passing Systems

As discussed, MPI applications that perform asynchronous message passing rely on sufficient amount of message buffers to prevent deadlock. Thus, the behavior of the communication depends upon the availability of message buffers. All asynchronous calls will turn out to be synchronous if the buffer resources are exhausted. Unfortunately both determining the minimum amount of buffers (The Buffer Allocation Problem) and determining whether an application may deadlock for a given number of buffers are intractable optimization problems [3]. An epoch based approach is developed by [4] to approximate the buffer allocation problem. In this approach, the execution of the application is partitioned into epochs, and after each epoch barrier synchronization is performed. It ensures that a process upon reaching the end of epoch will wait for every other process to reach the end of the epoch before it can proceed. As every process synchronizes at the end of the epoch, the buffer requirements needed to perform safe asynchronous message passing are considerably reduced. While constructing the graph in next chapter we will be using a part of BAP algorithm in [4] to compute the epochs for display.

Communication Graph

In order to define a communication graph we use the following definitions from [3]. A program trace S is a log of all events that occurred during an execution. Events are either a start of a process or completion of a send or receive operation. A communication graph G of a program trace S is a directed acyclic graph $G=G(S)=(V,A)$ where the set of vertices $V = \{v_{i,c} | 1 \leq i \leq n,\ 0 \leq c \leq e_i\}$ corresponds to events in the

trace, where e_i is the number of events performed by process i. Vertex $v_{i,0}$ represents the start event of process i and vertex $v_{i,c}$ represents either a send or a receive event. The former is called a start vertex and the latter are called send and receive events respectively. For each vertex $v_{i,c}$, i is called the process number and c is called the event number.

A communication arc $(v_{i,c}, v_{j,d}) \in C$ represents a communication between different processes, i and j, where $v_{i,c}$ is a send vertex and $v_{j,d}$ is a receive vertex (see Figure 2-2). The vertex $v_{i,c-1}$ is called the parent vertex of the vertex $v_{i,c}$, and the vertex $v_{i,c+1}$ is called the child vertex of $v_{i,c}$. The vertex $v_{j,d}$ connected to $v_{i,c}$ by a communication arc is called the sibling vertex of $v_{i,c}$. The communication graph contains an ordering of all events in the trace. That is, a path from vertex v_a to vertex v_b in the graph indicates that event a must occur before event b.

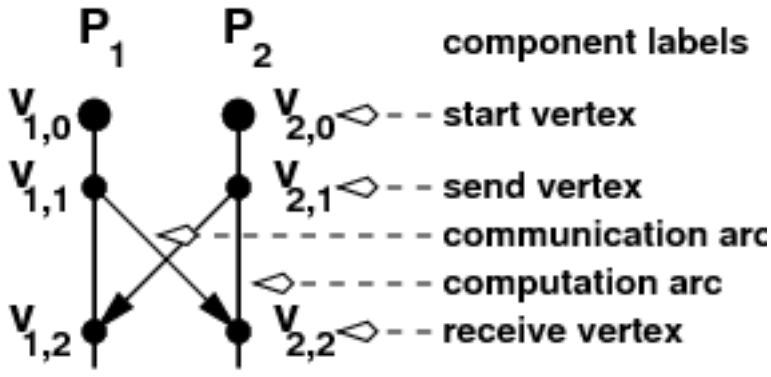

Figure 2-2: A communication graph for two processes.

An epoch E, which is a sub graph of G, is a self-contained sequence of consecutive events in the trace. The epoch E contains atleast one send or receive event. Since there can be no vertex outside the subgraph that has a path to and from a vertex in the subgraph, all epochs in G must be disjoint. Every epoch E has at least one send and receive vertex. An epoch is called simple if it contains exactly one send and receive vertex. An epoch is called complex if it contains more than two vertices. As stated in [4], a graph can be decomposed into a set of disjoint Epochs $G = E_0 \circ E_1 \ldots\ldots \circ E_n$ as shown in Figure 2-3.

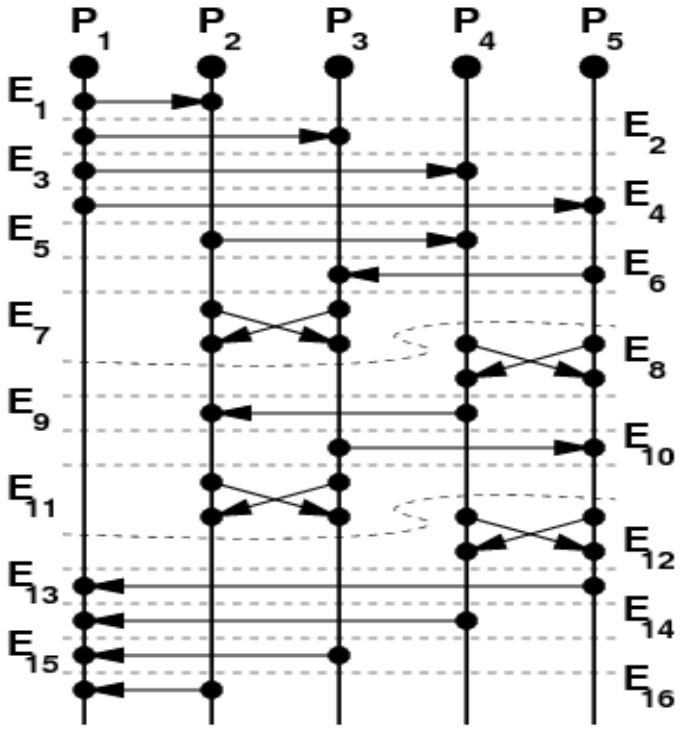

Figure 2-3: The graph G partitioned into epochs.

Protocol Conformance Checking

A protocol constraint specification contains a list of constraints that specifies a protocol for a given message passing system. The idea is to allow the user to specify the relationships between processes through constraints on the message passing and have the system check the messages against these constraints. It can be used in debugging and development cycles of parallel programming and also serves as a specification that can be used for testing purposes. One of the advantages of using protocol constraints is that, even if a protocol is not formally verified, it can be used in testing the implementation of the protocol. This could help in correcting the errors in the implementation which might otherwise be difficult to find at later stages. Protocol constraint specification is different concept when compared to constraint programming. The main goal of constraint system is to help in connection with the program development strategy. A program written in a constraint programming consists of a set of equations that are given to a constraint-satisfaction system which in turn, returns the values satisfying the constraints whereas protocol constraint specification does not generate any messages that satisfy the constraint system instead it uses message information to validate the constraints. Most tools that are used for protocol checking and verification require the protocol / model to be specified or implemented separately in the language of the tool, forcing them to re-implement the protocol in the same language that the application is written in.

A well known approach to perform protocol verification and checking, includes the process algebra CSP (Hoare's Communicating Sequential Processes) [19] developed at Oxford University. CSP is a formal language for describing patterns of interactions where processes proceed from one state to another by engaging in events. FDR (Failures-

Divergence-Refinement) [20] is a model checking tool used to test specifications written in CSP.

The inspiration for the tool described in next Chapter was obtained from [7]. A brief overview of [7] is given below. A protocol specification consists of a number of lines that specify which sends can send to which receives. A constraint would take the following form:

$$pgname_1[e_1]\{e_2\}(e_3) \rightarrow pgname_2\ [e_4]\{e_5\}(e_6)$$

The constraint line can be followed by a number of quantifiers which takes the following form:

$$\forall id:RelExpression;$$

The first part states that a process from program $pgname_1$ with instance number e_2 and send line no e_3 in group e_1 can send to a process $pgname_2$ in group e_4 with instance number e_5 and receive line no e_6. e_1, e_2 and e_3 can be omitted or be a number or an identifier and e_4, e_5 and e_6 can either be expressions, identifiers, or be omitted. If omitted, a wildcard match is performed. A quantifier introduces constraints on an identifier used in the e_1,….,e_6. These can be qualified by both lower and upper bound or bound by other expressions. A message sent from a sender to receiver is a tuple as follows:

$$M = (P_s, P_r, (G_s, I_s, L_s), (G_r, I_r, L_r), N_s, N_r)$$

Where P_s, G_s, I_s, L_s denotes the program name, group, instance and line of the sender, and P_r, G_r, I_r, L_r denotes those of receiver. N_s, N_r are the total number of processes in group G_s and G_r. We make direct use of this idea to implement the protocol conformance checking which is explained in chapter 3.

The above mentioned protocol specification can also be checked using different levels of strictness as shown in Table 2.1.

Level	Description
1	0 or more protocol specification lines may match with respect to program name and sender quantifiers.
2	At least one protocol specification line must match with respect to program name and sender quantifiers.
3	Exactly one protocol specification line must match with respect to program name and sender quantifiers.

Table 2.1: Strictness levels.

We also make use of strictness levels, but in a slightly changed way which we will see in the next chapter.

CHAPTER 3

MESSAGE PASSING AND PROTOCOL CONFORMANCE CHECKER VISUALIZATION TOOL (MPROV)

MPROV encompasses two parts: A standalone Java program which visualizes the communication patterns occurring in an MPI application and a protocol evaluator which validates the messages based on user defined constraints. MPROV needs two inputs: a log file describing the communication pattern of the message passing; such as source, destination, send or receive event numbers along with corresponding line numbers of the actual explicit message passing calls in the source code, and a protocol specifications file containing a list of constraints that the messages should satisfy. An `MPI` run creates a log file of the message passing for each process components; these individual log files are concatenated to form a log file for the entire execution. This log file is given as input to the Java program (see section MPI-Message Capture on how these are captured). It then uses this information to create the communication graph (based on Lamport's space time diagram [5]), which is then displayed on the screen. The graph also contains epochs (see section communication graph in chapter 2) that partitions the application's execution. The log file contains a number of entries as follows:

$$S_{id}:E=S_{no}:S:D_{id}:L_s\ ;$$

$$D_{id}:E=R_{no}:R:S_{id}:L_r\ ;$$

S_{id} denotes the id of the source of the message, D_{id} denotes the destination, S_{no}, R_{no} denote the corresponding send or receive event numbers along with line numbers L_s and L_r. Each send event must have a corresponding receive event in order to correctly construct the graph. `S` or `R` identifies a given message as a send or receive.

The tool was coded using Java Swing which provides a rich graphics functionality using object oriented programming. The Graphical User Interface (GUI) was designed using Netbeans 6.5. MPROV contains 3 main packages: A `GUI` package that contains classes which are used for drawing the graph, a `UI_Interaction` package that defines classes used to perform UI interactions and finally a `Protocol_Conformance` package that is responsible for protocol evaluation and constraint checking.

The log file is interactively given to the `String_Tokens` class that splits the entries in the log file into tokens which are sent to the `Vertex Class`. The `Vertex class` defines the entire data structure that includes source, destination, event numbers, line numbers and event type (a Boolean value specifying whether it is a send or receive). The `BAP` class uses this list of vertices to effectively partition the graph into epochs. It provides the communication pattern (source, destination, event numbers, epoch no etc) to the `Draw_CommunicationArc` class which calculates the send and receive coordinates to draw vertices (nodes) and edges for all process components. The entire procedure for converting the log file into a graphical representation is shown in Figure 3-1.

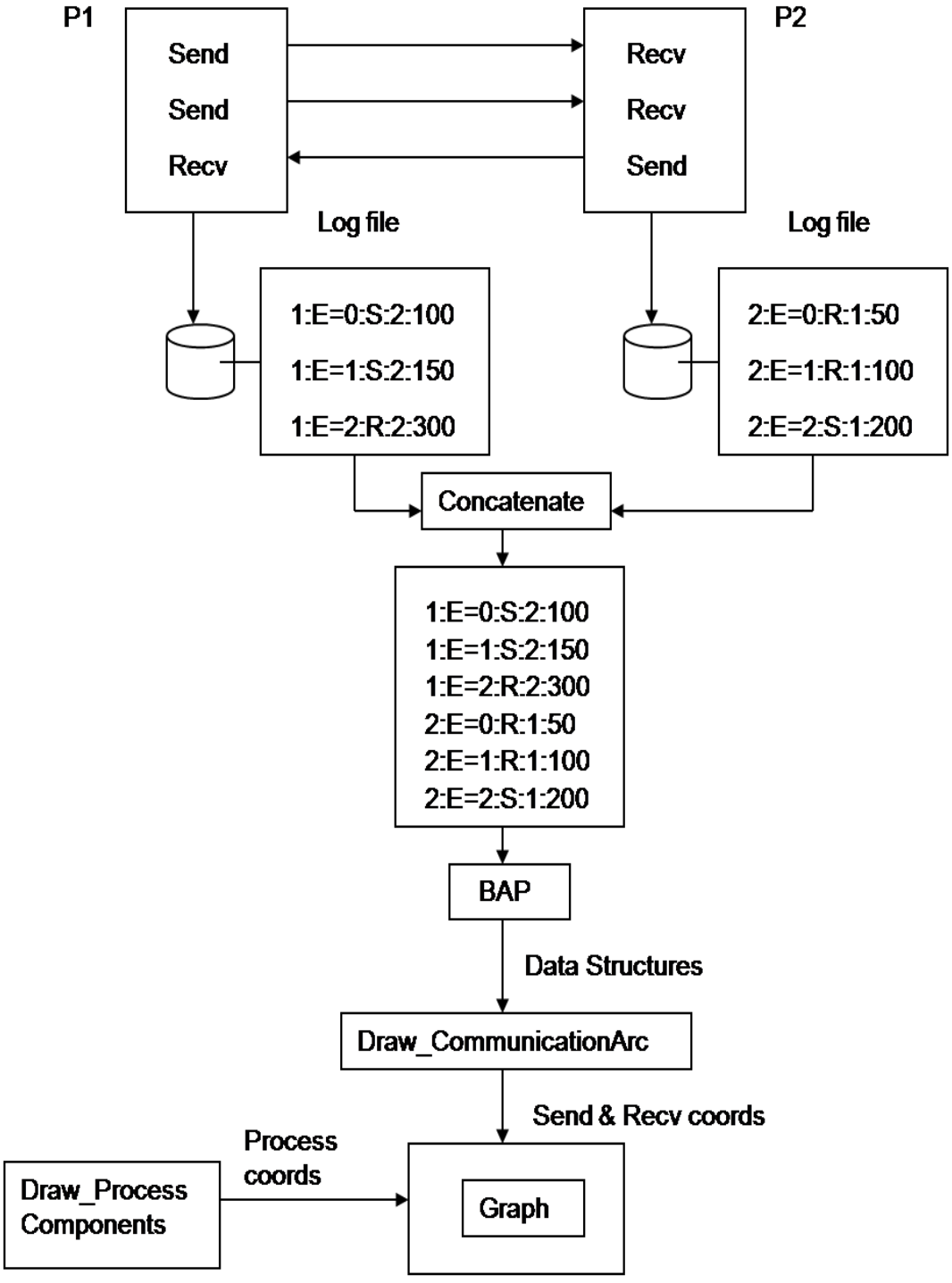

Figure 3-1: Illustration of Graph Drawing.

MPI–Message Capture

To create a log file, the user must execute the MPI application with a C-library that performs the data collection. The C-library will record every send and receive event in separate files for every processes. These individual log files are concatenated to form a single log file with a shell script. The c-code needed for collection is contained in a file called bap.c which acts as an interface between MPI program and the standard MPI library. The following MPI changes are done to capture the messages. The functions: `MPI_Init()`, `MPI_Send()`, `MPI_Recv()` and `MPI_Finalize` defined in the MPI library are redefined in bap.c that acts as wrappers around the original functions. Then we could use C-preprocessor *#define* macros to replace MPI calls with calls to the wrapper functions.

Graph Visualization

MPROV is used to visualize the message passing of an MPI program, and to analyze the correctness of the communication pattern. Consider a log file with the following entries that corresponds to the graph shown in Figure 3-2:

```
0:E=0:S:1:1;
1:E=0:R:0:1;
1:E=1:R:2:2;
2:E=0:S:1:2;
```

A vertical line in the graph signifies the processes (e.g. P_0, P_1, and P_2) and the edge connecting the process components correspond to the message passing taking place

between the processes. As shown in Figure 3-2 the numerical representation (0, 1) signifies that process P_2 at event number 0 is sending a message to process P_1 at event number 1. E_1, E_2 are the corresponding epoch numbers.

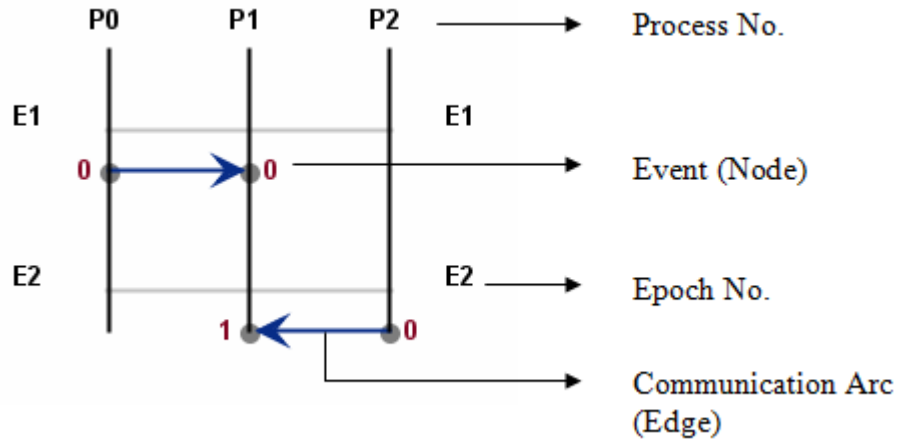

Figure 3-2: Graphical Representation of Communication Pattern.

The `DrawCanvas_UI` class

The `DrawCanvas_UI` is the largest class that implements the UI objects. The `DrawCanvas_UI` creates a `JFrame` which is the main container of the GUI. The `JFrame` is divided into two sections, namely the drawing area located at the center and an interaction panel placed on the right hand side as shown in Figure 3-3. The interaction panel contains interaction techniques that can be applied to the graph. It also contains a strictness level panel containing option buttons to apply the desired strictness level once the protocol specification file is loaded. The `DrawCanvas_UI` also contains a sub-class `DrawCanvas` which creates a custom `JPanel` to perform custom painting. Then, the custom `JPanel` is placed over the `JScrollPane` to implement a scrollable interface.

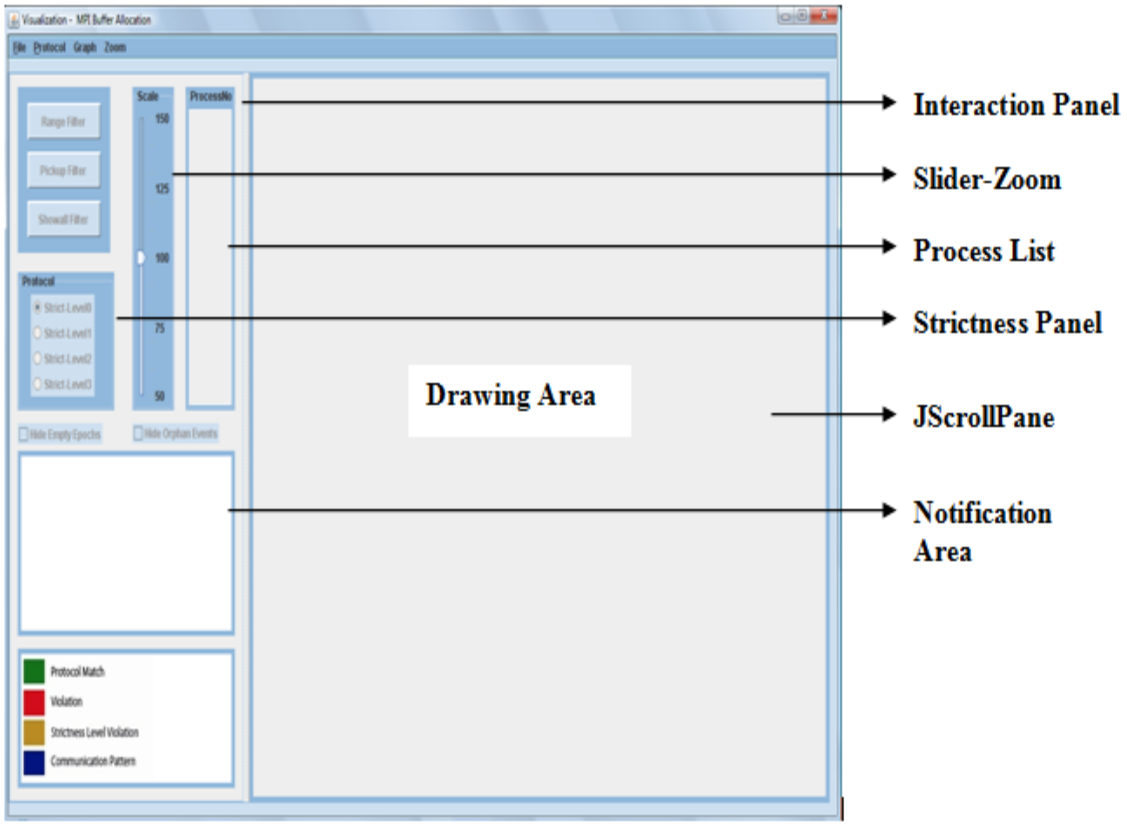

Figure 3-3: The Layout of the Interface.

The log file is loaded into the Java program using the file menu item as shown in Figure 3-4.

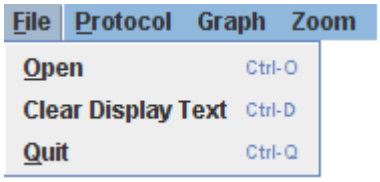

Figure 3-4: File-Open Menu.

The different Menu Bar options and their uses are depicted in Table 3-1.

Menu	Menu Item	Description
File	Open	To load the log file and draw the communication graph.
File	Clear Display Text	To clear the notification area.
Protocol	Load File	To load the protocol specification file.
Protocol	Reload File	To apply the updated specification to the existing protocol file.
Protocol	View File	To view protocol constraints.
Protocol	Apply	To apply the protocol conformance checking.
Protocol	View Results	To view the results after checking the constraints.
Graph	Reset	To reload the original graph.
Zoom	Reset Zoom	To reset the zoom to original setting.

Table 3-1: Menu bar functionality.

Graph Drawing

The co-ordinates to draw the edges that denote the message flow are calculated using `g2d.drawline(`x_1`,`y_1`,`x_2`,`y_2`)` where `g2d` is the Graphics2D object (a subclass of Graphics) that contains a richer set of drawing and display operations to perform advanced 2D graphics. Coordinates (x_1,y_1) and (x_2,y_2) are the start and end points of

the line (edge) that connects the nodes, where x_1 and x_2 corresponds to the x coordinates of the send and receive process components respectively. This calculation (x_1 and x_2) is done in the `Draw_ProcessComponents` class using `getVertline_Xcoord (int processno)` method. The `Draw_CommunicationArc` class contains methods to calculate y_1 and y_2. The algorithm to calculate y_1 and y_2 is shown below:

Algorithm: Calculate Send and Receive coordinates.

Input: source, destination, send event, receive event, epoch no.

Output: An arraylist containing sends and receives coordinates (y_1 and y_2).

```
Step1:previous_epoch = 0;

Step2:For each src, dest, sendevt, recvevt, epoch_no

Step3:if (epoch_no > previous_epoch)

/* epoch coord is calculated when epochno is different
 * maxcoord is max (send or receive coord)
 */
           previous_epoch += 1;
           epoch_coord = maxcoord + DELTA_Y;

Step4:newsendEvent = max_y_value + sendevent - sendEvtmax;

Step5:send_coord = INITIAL_COORD + (newsendEvent*DELTA_Y) + (Epoch_no*DELTA_Y);

/* Where INITIAL_COORD & DELTA_Y are constants */

Step6:newrecvEvent = max_y_value + recvevent - recvEvtmax;

Step7:recv_coord = INITIAL_COORD + (newrecvEvent*DELTA_Y) + (Epoch_no*DELTA_Y);

Step8:Add send_coord and recv_coord to corresponding arraylist.

Step9:Next;
```

Functionality of GUI Components

The ability to visualize and to test slightly different graphical views may help understand and verify the correctness of the communication patterns. On visualizing a large graph a good layout algorithm alone will not solve the problem as the visual

representation of the graph could be very complex. Limiting the number of visual elements to be displayed could be one possible solution to improve clarity. Clustering functions like filters which display only the processes of interest in the graph prove to be useful while displaying large graphs. MPROV provides two filters: A range filter, which restricts the number of processes displayed on the screen. The user is prompted to enter the range (start process and end process) of the processes of interest, and the graph is filtered to display only processes which fall within the specified range.

The pickup filter is used to randomly pick a set of processes**.** The user has to select more than one process from a list box and click the "Pickup Filter" button. Multiple Processes can be selected using the CTRL key. The pickup filter has also been implemented using a similar algorithm as the range filter. The need for the pickup filter arises when the total number of processes is large. For example, if there are 300 processes, and the user is interested in 4 processes that are widely spaced (example, P_4, P_{120}, P_{250}, P_{300}); then a pickup filter may be used. On the other hand, a range filter is used when the user is interested in viewing the communication between a range of consecutive processes. Theoretically, a pickup filter may be used instead of a range filter by individually selecting the processes of interest over a given range. Though, a pickup filter may mimic a range filter, both of them individually add value to the tool.

Most often range and pickup filter fills the graph with empty epochs and orphan events. An empty epoch occurs when no communication takes place between the selected processes in that epoch. Orphan events may be a send or receive orphan event. A send orphan event occurs when a matching receive event is missing and receive orphan event occurs when there is no matching send event. This scenario may occur when the processes

are filtered and the corresponding send or receive events are not a part of the filtered components.

To illustrate the workings of a filter, Figure 3-5 contains 4 process components P_0,…,P_3 and 4 epochs E_1,…,E_4. When the pickup filter is applied to the graph with P_0 and P_1 as the processes of interest, the resulting filtered graph is as depicted in Figure 3-6.

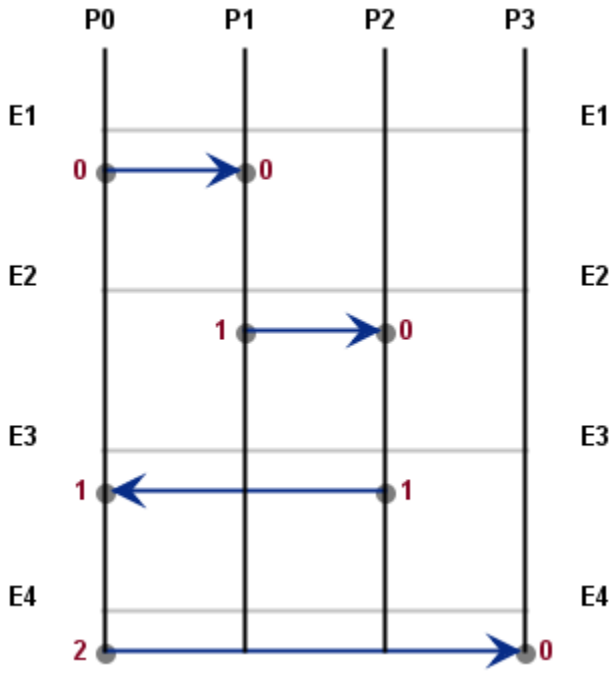

Figure 3-5: Graph with 4 process components.

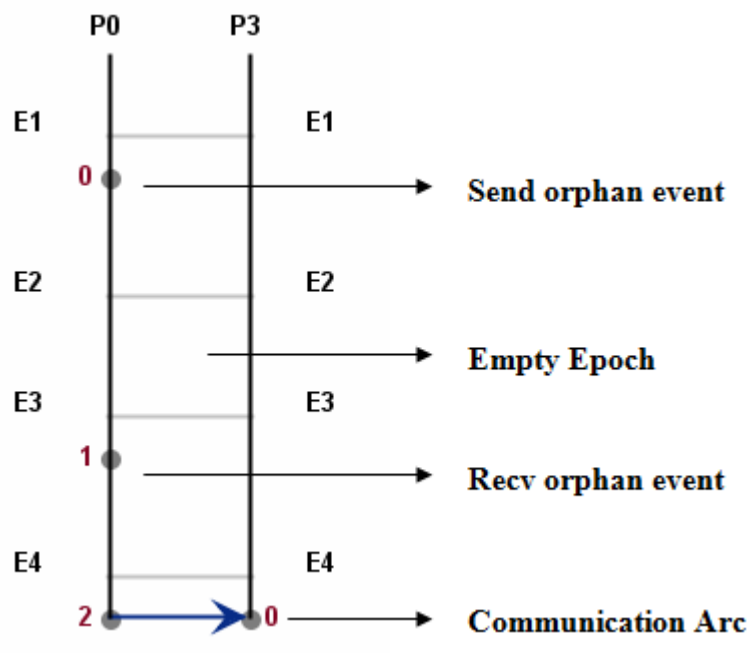

Figure 3-6: Filtering Process Components P_0 and P_3.

Hiding the orphan events and empty epochs provides us with multiple abstraction levels. The hide empty epochs and hide orphan events checkboxes hide the empty epochs and orphan events correspondingly. A brief description of the UI components is given in Table 3-2.

Name	Description
Range Filter	This button enables the user to specify a range of processes to be displayed.
Pickup Filter	This button enables the user to randomly select the processes to be displayed.
Showall Filter	This button is used in combination with the Pickup filter to display the entire graph. All the processes except those selected by pickup filter are displayed in shaded gray color.
Hide Empty Epochs	This checkbox enables the user to hide empty epochs.
Hide Orphan Events	This checkbox enables the user to hide orphan events.

Table 3-2: GUI Functionality.

Zoom

MPROV provides a zooming feature to analyze the program behavior on any level of detail. The zooming feature can also be used to get deeper and deeper into the analysis process, to not only understand program behavior, but also to verify the correctness of the communication patterns. The tool incorporates the slider zoom and the mouse wheel zoom. The slider zoom is integrated with a numerical value bounded by a minimum and maximum value. On moving the slider's knob, zoom in or zoom out operation are performed depending upon the numeric value selected. A minimum zoom level of 50 and a maximum of 150 have been provided. The mouse wheel zoom, zooms in and out relative to mouse position. The zoom value increases on moving the wheel downwards and decreases on moving upwards.

Protocol Constraint Specification

The protocol specification file is loaded into the application by using the `protocol → loadfile` menu item as shown in Figure 3-7.

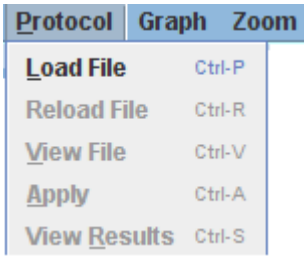

Figure: 3-7: Protocol-File Load Menu item.

A protocol consists of a number of constraint lines of the following form:

$$P(e_1,e_2) \rightarrow P(e_3,e_4)\ [::Q];$$

The first part states that process with instance number e_1 and send line number e_2 can send to a process with instance number e_3 and receive line number e_4. e_1, e_2, e_3, e_4 can be empty or a wildcard *, a number or an identifier. Both empty and a * signifies a wildcard match. Q is a set of quantifier that introduces constraints which are applied on the identifier. It can be qualified by both upper and lower bounds. The quantifier Q can be represented by the following form:

$$\forall id: Relational\ Expression;$$

Example: $P(r,*) \rightarrow P(r,*)::\forall r{:}r == 1;$

Protocol Constraints Transformation

Once the protocol specification file is loaded, `protocol → apply` menu item may be selected to apply the protocol conformance check. Since e_1 and e_2 can be either empty (*) or a number (constant) or an identifier, a conversion, which we refer to as an α-conversion is done in order to bring all sender parts of a protocol line into a canonical form as given below:

$$P(\alpha_1,\alpha_2) \rightarrow P(e_3,e_4) :: \forall\ \alpha_1{:}xxx;\ \forall\ \alpha_2{:}xxx;\ Q'$$

The following algorithm defines how α-Conversion is done.

Algorithm: α-Conversion

- If e_i is a number (c_i), e_i is replaced by α_i, and the quantifier $\forall \alpha_i{:}\alpha_i{=}c_i$ is added to Q.
- If e_i is an identifier, replace its entire occurrence by α_i.
- If e_i is * or empty replace it with α_i, and add quantifier $\forall \alpha_i$: true to Q.

The above transformation is done so that the sender part of the protocol line can be checked separately from the rest of the quantifiers and if they don't satisfy we don't have to check the receiver part. The γ-Conversion is done on the receiver side to bring the protocol lines into one of the forms:

$$P(\alpha_1, \alpha_2) \rightarrow P(e_3, e_4) :: Q;$$
$$P(\alpha_1, \alpha_2) \rightarrow P(e_3, \gamma_4) :: Q;$$
$$P(\alpha_1, \alpha_2) \rightarrow P(\gamma_3, e_4) :: Q;$$
$$P(\alpha_1, \alpha_2) \rightarrow P(\gamma_3, \gamma_4) :: Q;$$

The following algorithm defines how γ-Conversion is done.

Algorithm: γ-Conversion

- For e_3 and e_4 the following transformation is done. If e_i is *, replace e_i with γ_i and add the quantifier $\forall \gamma_i$: true to Q.

- The above transformation is done in order to avoid comparing numbers with empty expressions.

The conversions are done only once at load time. Once the α and γ conversions are done, then we can perform checking to make sure that the values are within the boundaries of the definitions. Before any checking can be performed, we have to add information from Message `M` to symbol table. A Symbol table is used to associate variables with values. Each message `M` satisfies a protocol specification line `L`:

$$P(e_1,e_2) \rightarrow P(e_3,e_4) :: Q;$$

if and only if, algorithm `Protocol Constraints Validation` returns true.

Where $M = [S_{id}, S_{line}, R_{id}, R_{line}]$ is a message from the execution (that is, an arc from the graph), `Q` is the list of quantifiers.

Before we start evaluation, we create two arraylists `L_list` and `A_list`. All the messages that violate the protocol lines are added to the `L_list` and those which match are added to the `A_list`. For each message `M`, add to the symbol table with bindings $\alpha_1 = S_{id}$ and $\alpha_2 = S_{line}$. We continue evaluation, only if the sender part of the message satisfies the protocol line. If the check fails, then the message along with protocol line is added to `L_list` and the message is checked against the next protocol line. Once the evaluation returns true, then we need to check if the receiver part of `M` matches the receiver part of the protocol line. If the receiver parts are identifiers then they are placed into the symbol table with corresponding R_{id} and R_{line} respectively. The receiver part is now evaluated. If it satisfies, then they enter into the `A_list` and if false then some violations have occurred and it enters into `L_list`.

Strictness Level Color Coding

Colors are used to represent different kind of activities in the graph and to group related items to direct or command attention. Blue color denotes communication between process components. To color the graph after applying the protocol constraints we use the same arraylists `A_list` and `L_list` which contains the matched and the violations list as described earlier. `Check_Violmsg(String msg)` method determines whether a message is in the violation list or not. `msg` contains the string representation of the message of the following form for example:

$$S_{id},\ R_{id},\ S_{evt}$$

where Sevt is the send event. The messages that violate the protocol specification are represented in red and those which satisfy the specification are colored green. A protocol specification can be checked using different levels of strictness.

Even if a message satisfies a protocol specification line, it might violate the strictness levels. Strictness level 0 is the default level. When the strictness level of 0 is applied, if a message violates a protocol line `i`, it must match at least one to avoid violating the protocol. Strictness Level of 1 can have 0 or matches with 0 violations. In case of strictness level of 2, at least one protocol specification line must match with respect to the sender along with 0 violations. Finally, when strictness level of 3 is applied, exactly one protocol specification line must match with respect to sender along with 0 violations. All Strictness level violations are depicted using orange color. Table 3-3 shows the color coding format for different strictness levels.

	| L | > 0	**| L | = 0**	
Level 0			**| A | = 0**
Level 0			**| A | > 0**
Level 1			**| A | = 0**
Level 1			**| A | > 0**
Level 2			**| A | = 0**
Level 2			**| A | > 0**
Level 3			**| A | = 0**
Level 3			**| A | > 1**
Level 3			**| A | = 1**

Table 3-3: Strictness level color format.

Example: Consider a graph with 5 process components $P_0,\ldots,P_4$ containing 6 epochs $E_1,\ldots,E_6$ as shown in Figure 3-8. On applying the following specification lines

```
P(i,*) → P(m,*):: ∀i:i!=4, ∀m:m==4 ;
P(3,*) → P(4,*) ;
P(0,*) → P(1,*) ;
```

communication arcs in the graph will have some color change. As strictness level of 0 is the default strictness level, the communication arc that matches the protocol lines are colored green, while those that violate the protocol lines are colored red as shown in Figure 3-9. When applying strictness level of 1, the communications arcs from P_0 to P_1

at event numbers (0, 2 and 2, 5) changes from green to orange since it violates the strictness level. This is illustrated in Figure 3-10. On applying strictness level of 2, the resulting graph is same as before as it does not violate the strictness level. On the other hand, when strictness level of 3 is applied, the communication arc from P_3 to P_4 is a strictness level violation as it matches more than one specification lines; hence its color is changed from green to orange as illustrated in Figure 3-11.

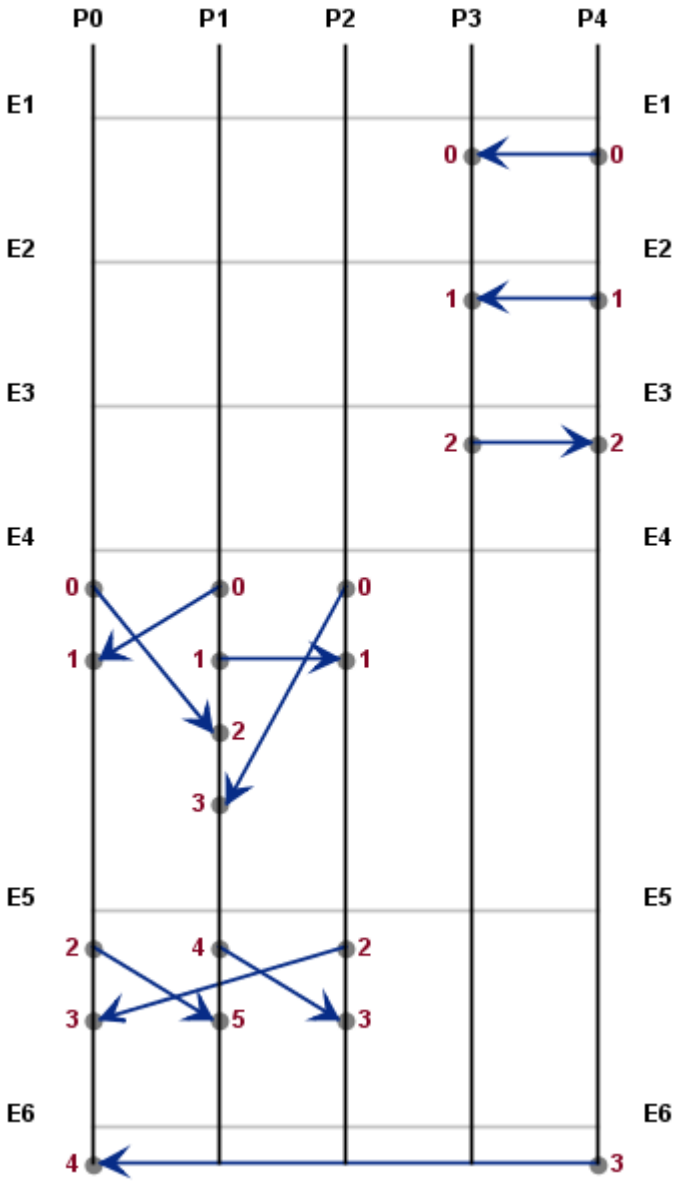

Figure 3-8: Before applying protocol specification file.

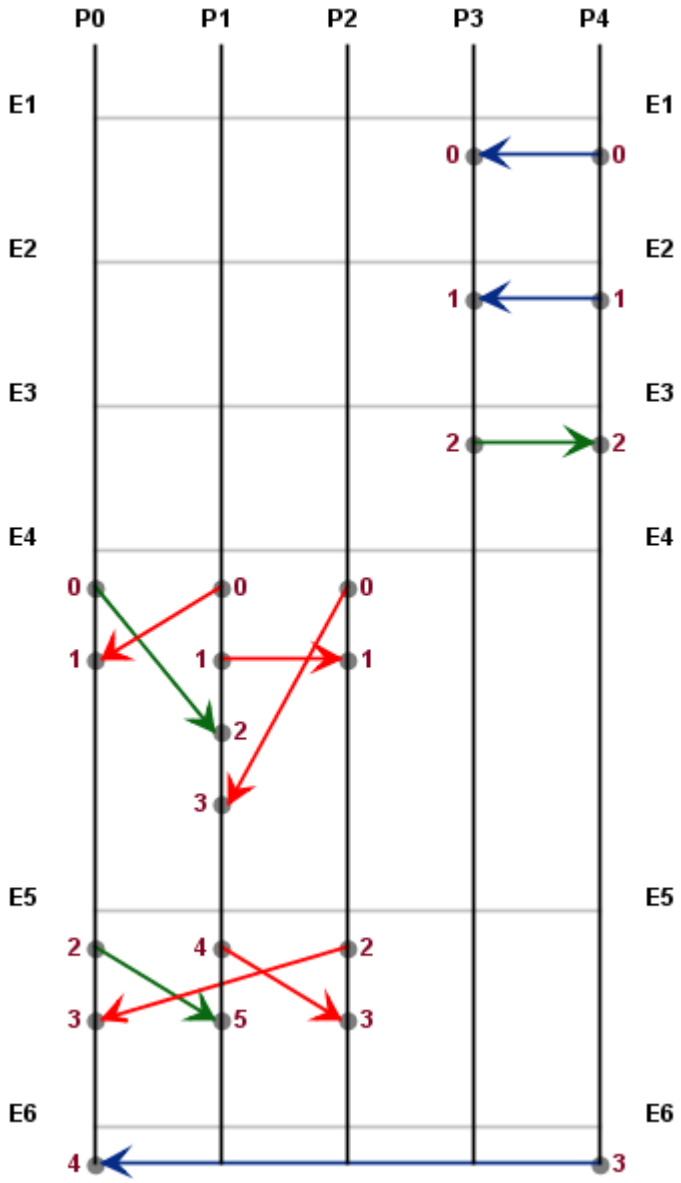

Figure 3-9: After applying strictness level 0.

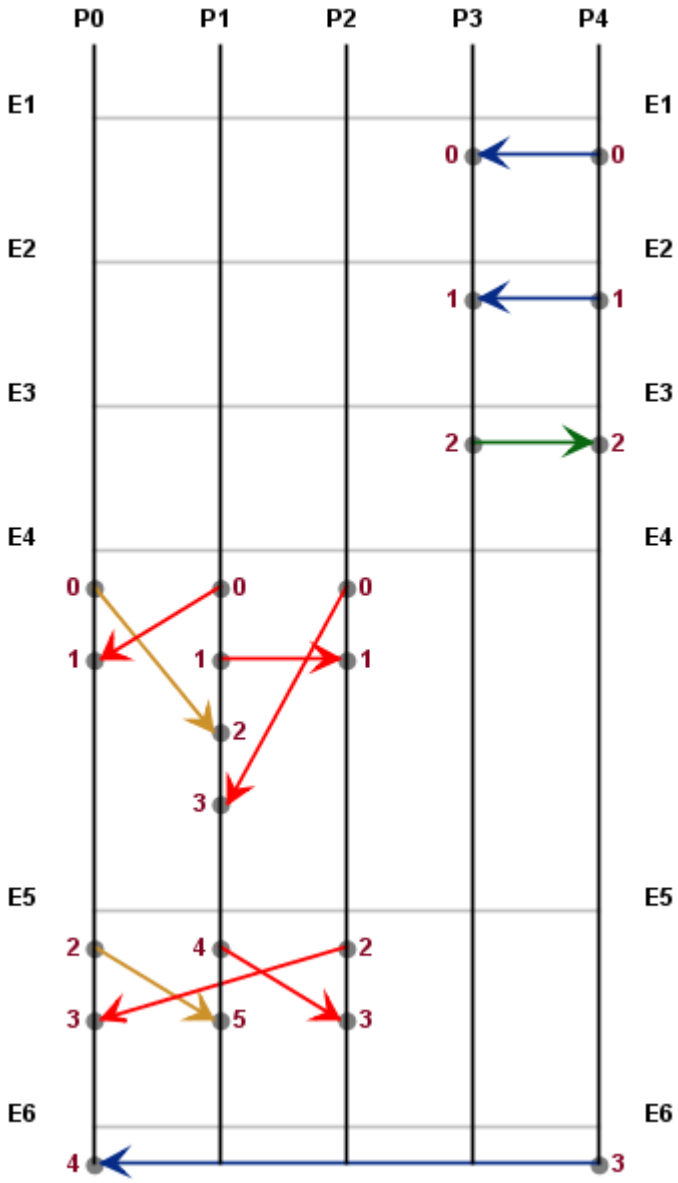

Figure 3-10: After applying strictness level 1 or 2.

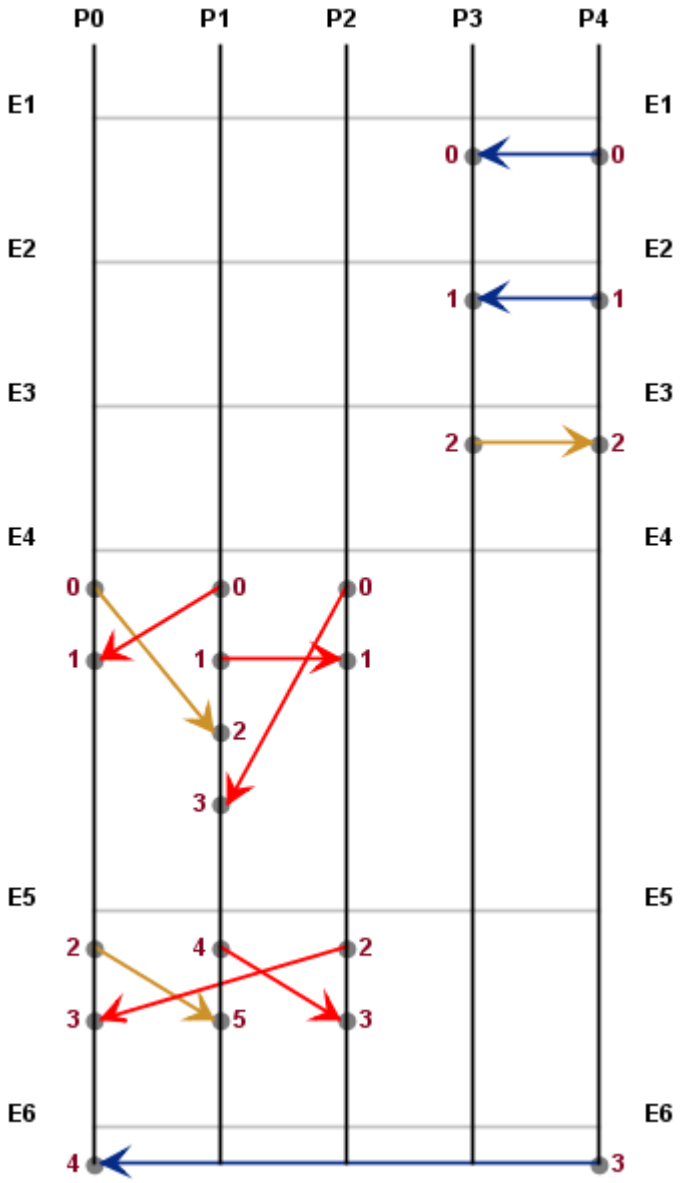

Figure 3-11: After applying strictness level 3.

Message Graph Checker

To check a protocol line L against a message M represented by send vertex, the following algorithm is performed:

Algorithm: Protocol Constraints validation

Input: M, L

Output: ‘true ‘if message satisfies the specification and ‘false’ otherwise.

For each Message M do the following

Step1: For each protocol line L do:

Step2: Insert into symbol table α_1 = Sid and α_2 = Sline.

Step3: For e_3,e_4 if they are identifiers insert into symbol table with values I_r, L_R.

Step4: Evaluate α Quantifiers. If α evaluated to true then protocol line matches the sender move on, else terminate and return false;

Step5: The expression e_3 is evaluated over the symbol table with the value I_r.

Step6: If (true) then e_4 is evaluated with L_R;

Step7: If (true) then M goes into matched list and return true;

Step8: else M goes into violation list and return false;

Step7: Next L;

Step8: if (M violates strictness levels) add M to report with corresponding strictness level;

Step9: Next M;

Node Level Tool Tip

Node level tool tip feature resembles a tool tip. This is a very useful feature to display information at the node level. When the user hovers the cursor over any send or orphan node, the cursor changes to the hand cursor, prompting the user for a click event. On clicking the node, a window pop’s up displaying details about the node such as source, destination, send or receive event no and epoch no. If a protocol specification has been

applied, the results displayed in the pop up window include the protocol lines. Figure 3-12 depicts the look and feel of the node level tool tip.

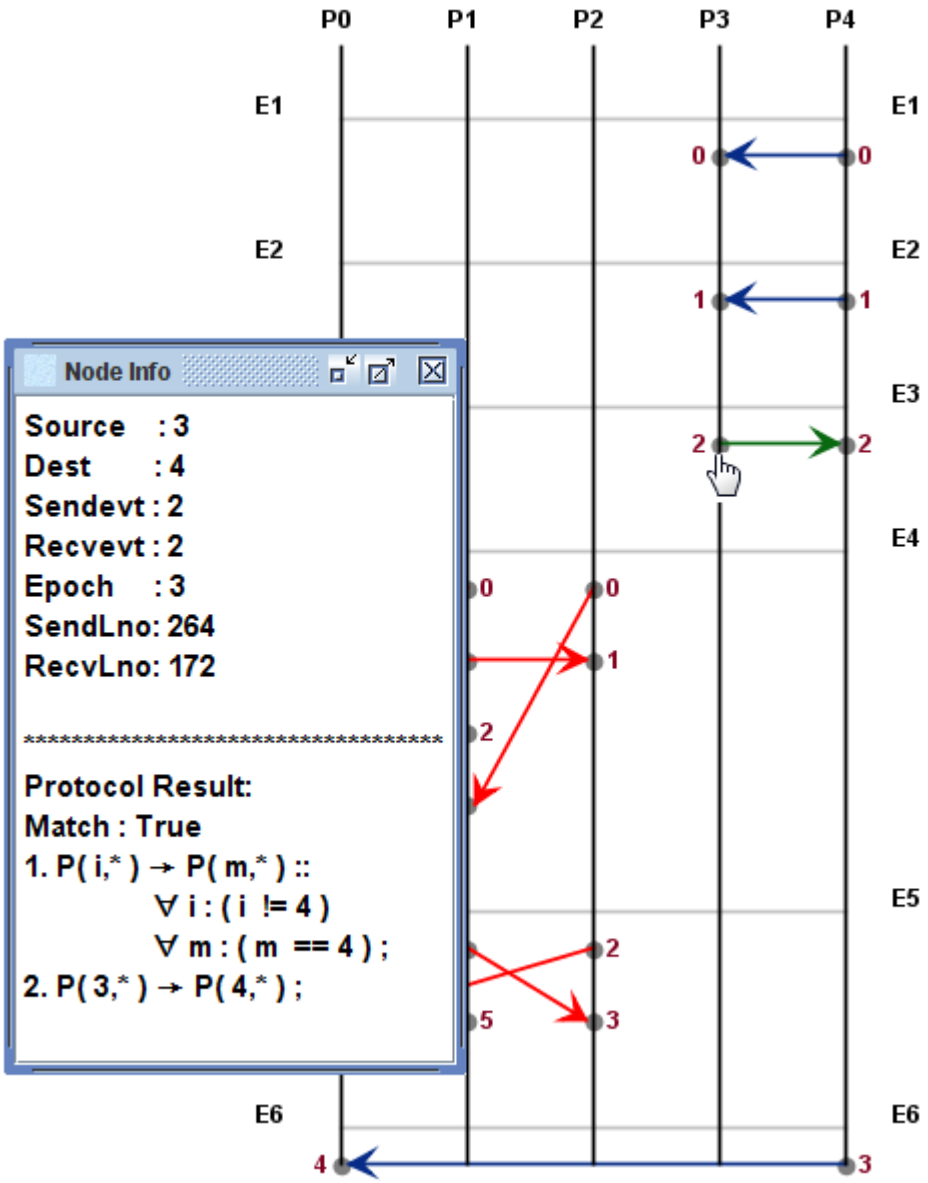

Figure 3-12: Node level tool tip.

CHAPTER 4

RESULTS AND SNAPSHOTS

In this chapter we discuss the results in detail and show several examples of using the tool. To enunciate the correctness of the visualization tool discussed in chapter 3, we use the following log (dataset) files: `dataset3.txt` containing 145 processes with 3,744 epochs and 7,488 vertices and `DES-10-collect.txt` containing 10 processes with 14 epochs and 178 vertices. A graphical representation of the communication between the 145 processes in `dataset3.txt` is shown in Figure 4-1a-d. Due to space constraints only a few snapshots of the application are shown. Also, in this chapter we have illustrated a few examples to demonstrate each feature of the tool.

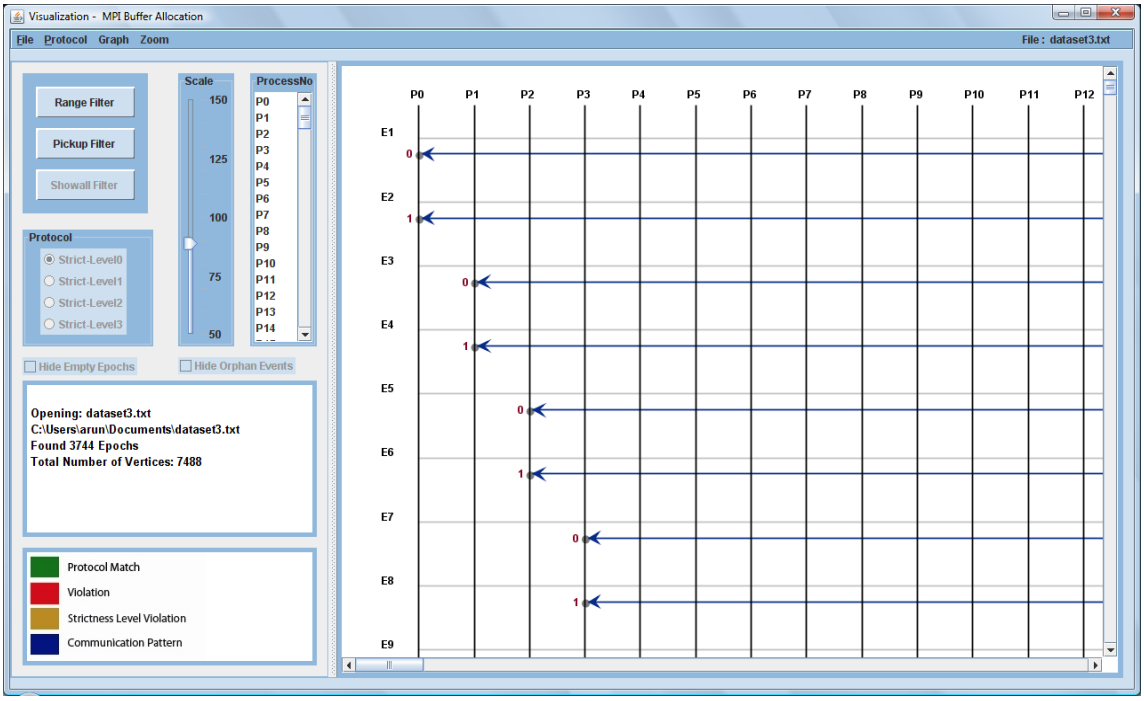

Figure 4-1a: A communication graph containing 145 processes (`dataset3.txt`).

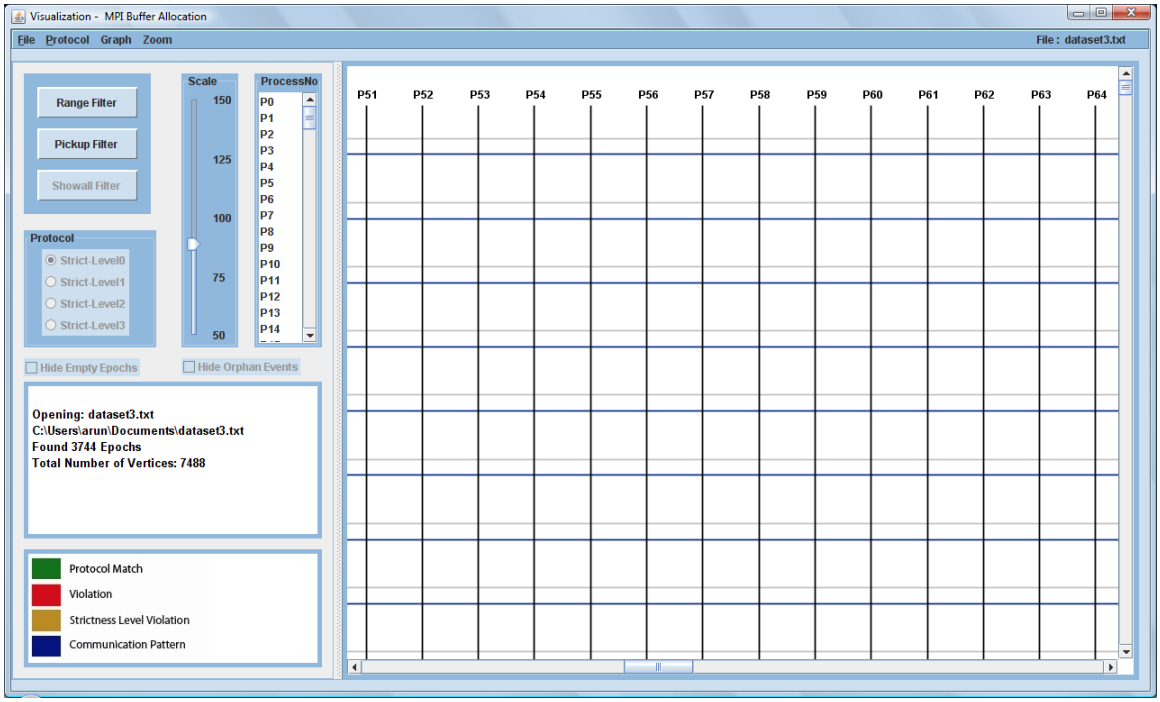

Figure 4-1b: A communication graph containing 145 processes (`dataset3.txt`).

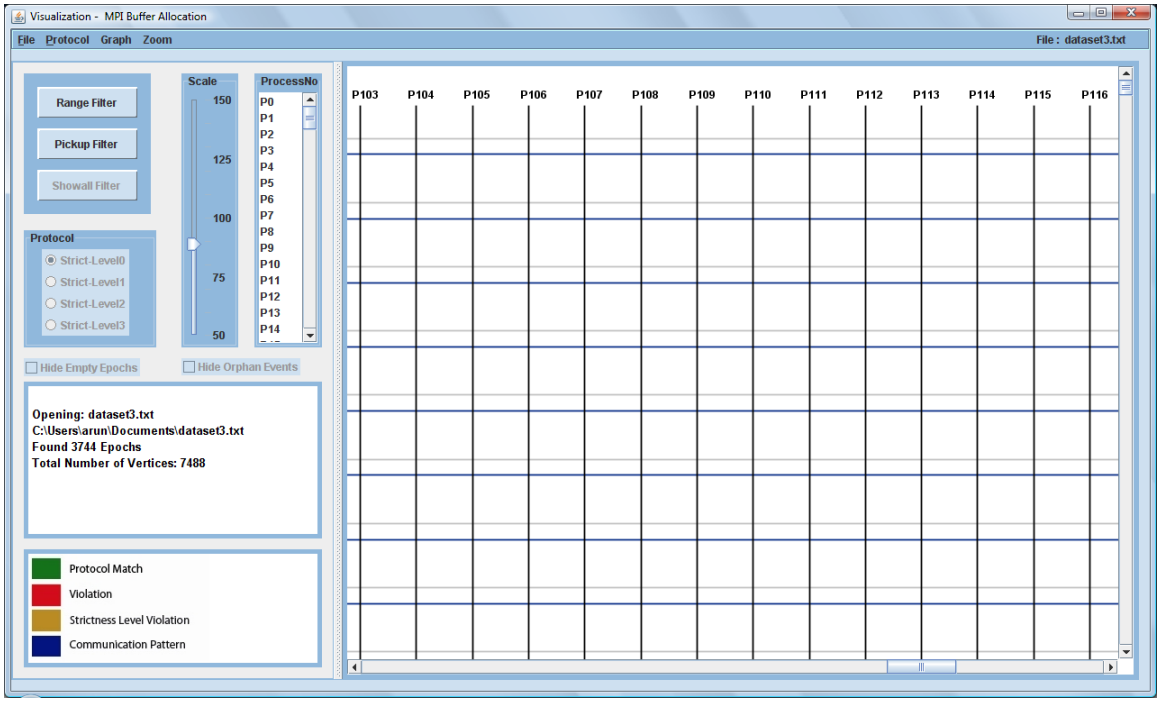

Figure 4-1c: A communication graph containing 145 processes (`dataset3.txt`).

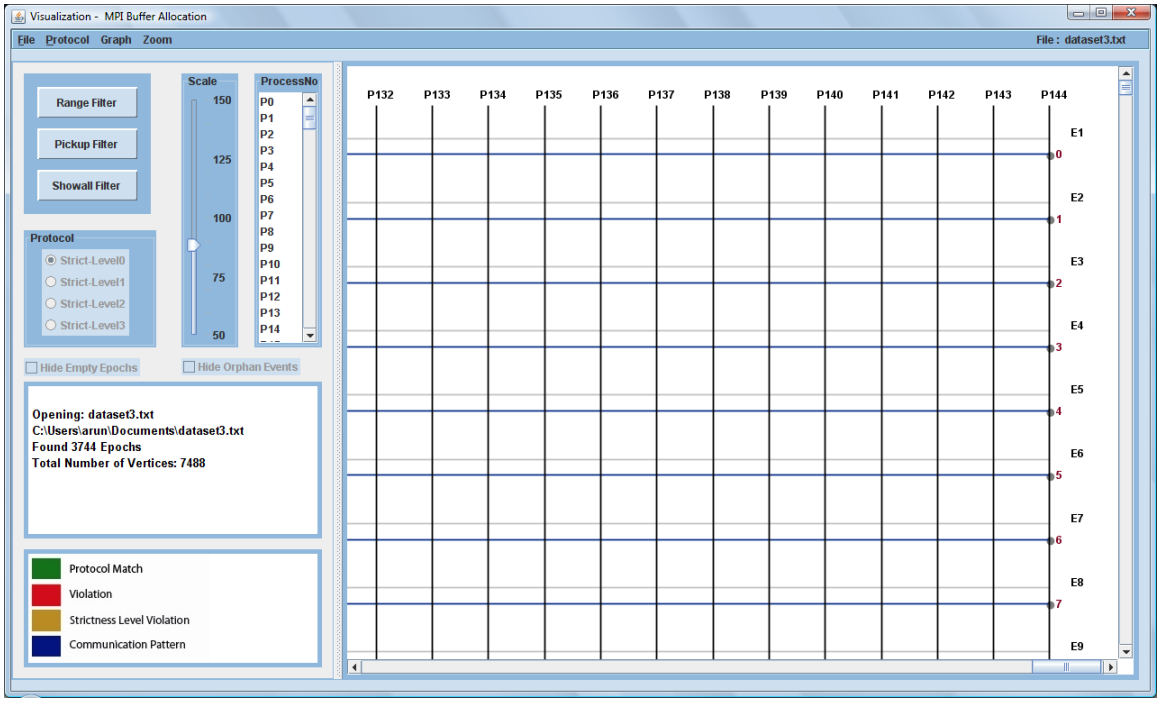

Figure 4-1d: A communication graph containing 145 processes (`dataset3.txt`).

Since we have not applied any protocol specification, all messages are colored blue. As any other visualization tool, interaction techniques are important; especially when dealing with large graphs. Filters are very useful in exploring large graphs and to satisfy the user's interest pertaining to a certain portion of the graph. We now discuss the results on applying the different filters to the graph.

Range Filter

On applying the range filter, the graph is filtered to display only those processes that fall within the requested range. We consider an example, where the start process is set to 100 and end process is set to 110. The dialogue box for entering the range is shown in Figure 4-2. Figure 4-3a and 4-3b shows the resulting graph after applying the range filter.

Examples of a few orphan events are as follows: event numbers 0 and 1 for P_{100} to P_{110}, event number 12 and 13 for P_{100} to P_{110}. Furthermore, E_1,....,E_{393}, E_{408}, E_{413}, E_{414}, E_{419}, E_{420}, E_{422},...,E_{424}, E_{426}, E_{430},...,E_{441} are examples of empty epochs obtained after applying the range filter.

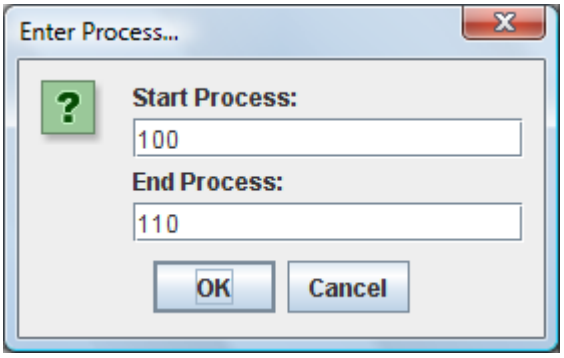

Figure 4-2: Range filter.

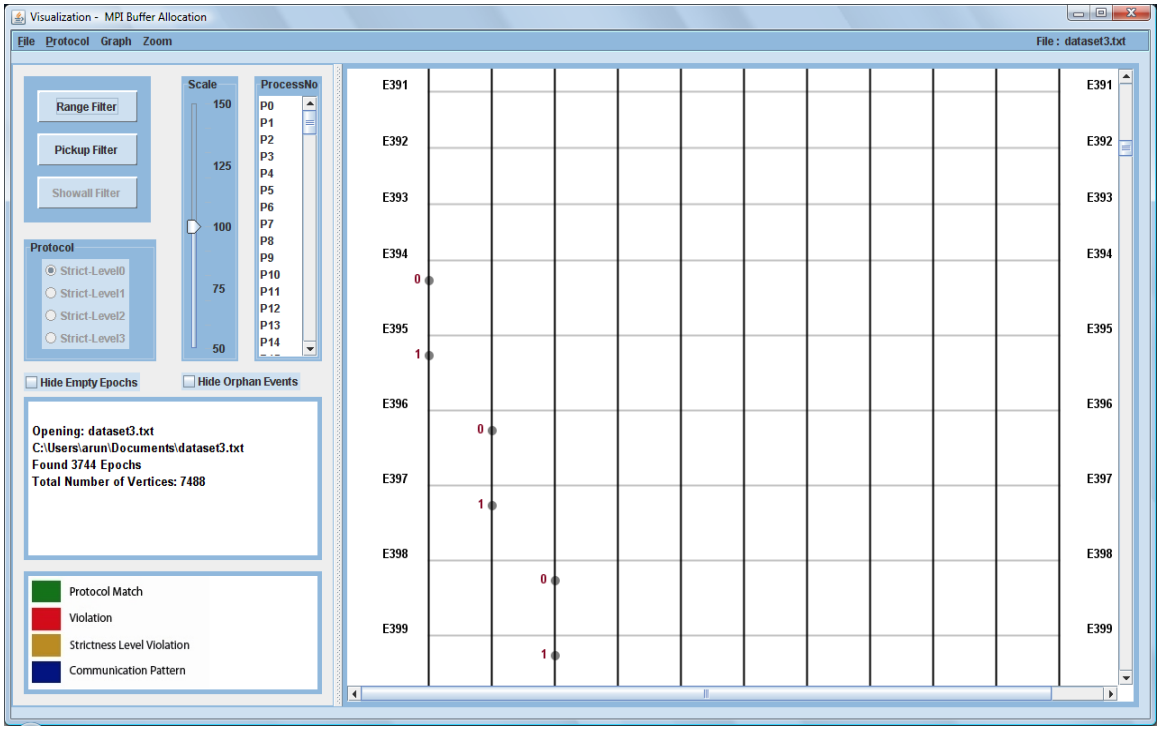

Figure 4-3a: Filtered graph after applying the range filter.

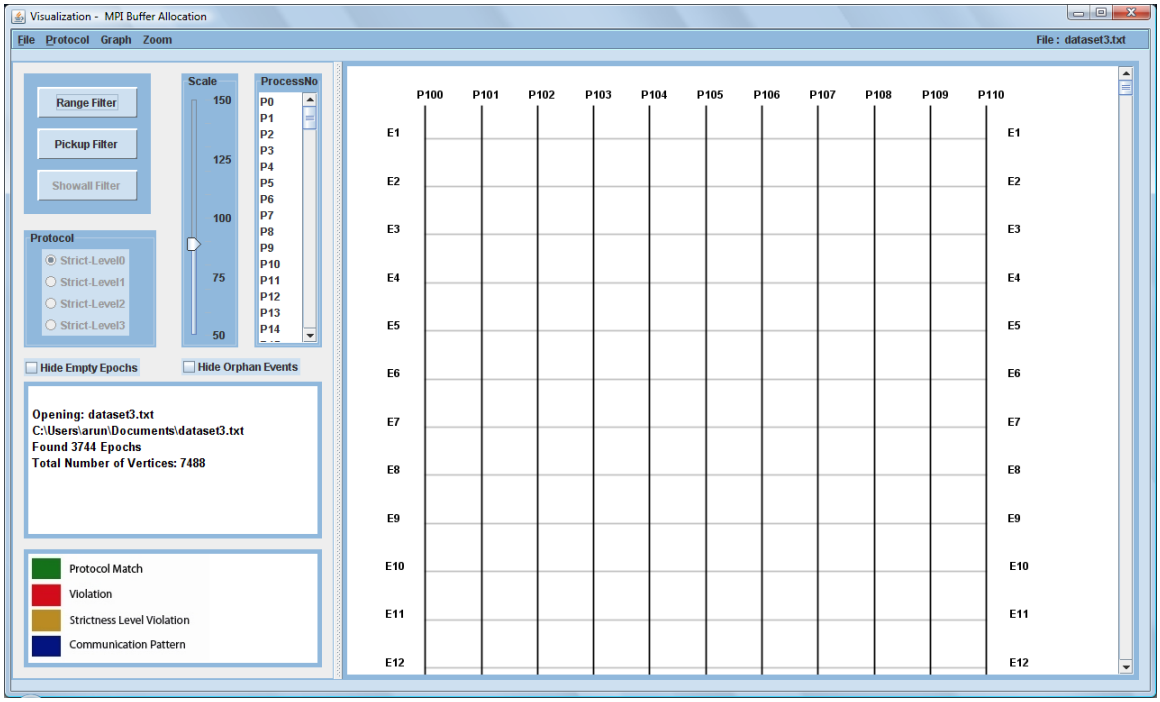

Figure 4-3b: Filtered graph after applying the range filter.

Pickup Filter

The pickup filter enables the user to select the process of interest from a given list of available processes. To select process of interest for the pickup filter, the UI presents a list box containing a list of processes as shown in Figure 4-4. Figure 4-5 shows the resulting graph after applying the pickup filter. In this example processes P_0, P_3, P_{10}, P_{83}, P_{87}, P_{102}, P_{134} and P_{144} are selected; hence, the filtered graph only displays the communication pattern between those processes. As depicted in Fig 4-5, the graph encompasses the following orphan events: 2, 3, 4, 5, 8, 9,....,19 for process P_{144}; 2, 3, 4 for process P_{102} and P_{83}; 2, 3, 4, 5 for process P_{87}, P_3 and P_{10}; 2, 3, 4 for process P_{102}.

Furthermore, E_{31}, E_{34}, E_{37}, E_{40}, E_{441}, E_{443}, E_{445}, E_{865}, E_{866}, E_{867}, E_{872}, E_{863},....,E_{868}, E_{870},....,E_{875}, E_{1628},....,E_{1635} and E_{2579},....,E_{2615} are examples of empty epochs.

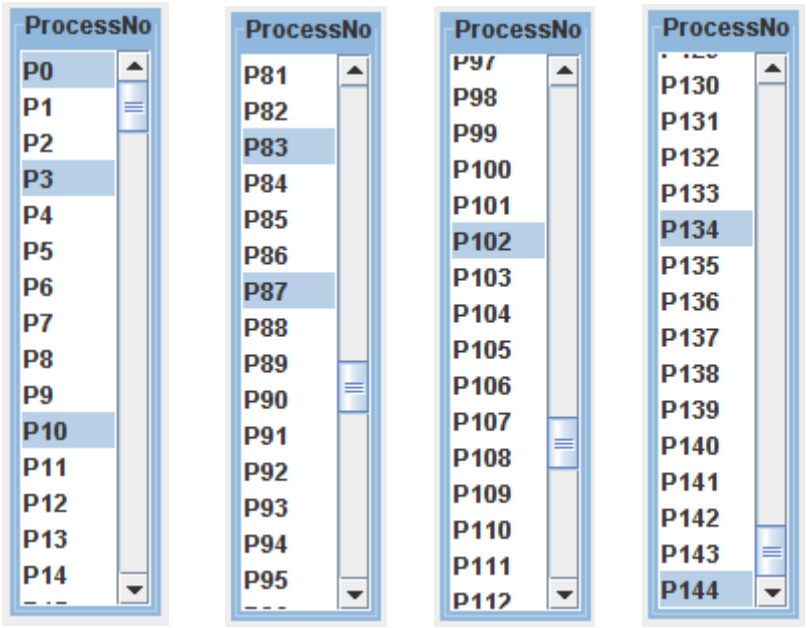

Figure 4-4: A list containing all processes for the pickup filter.

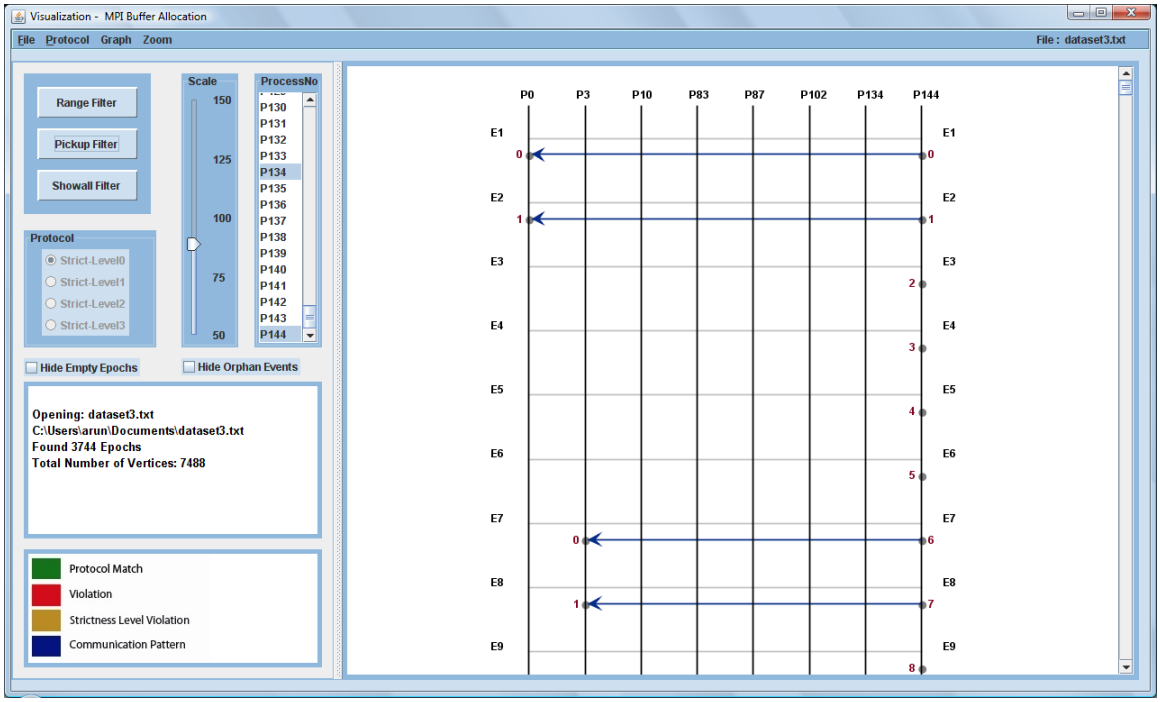

Figure 4-5: Graph after applying the pickup filter.

Showall Filter

The showall filter is designed to work in combination with the pickup filter. On applying the showall filter, the non-filtered processes of the graph and their corresponding communication arcs are displayed in a gray color. In other words, the user can view the entire graph with the filtered processes being highlighted and the non filtered process being grayed out. A snapshot of this feature is show in Figure: 4-6a and 4-6b. In this graph, process P_{144} at event number 0 is sending a message to process P_0 at event number 0; process P_{144} at event number 1 is sending a message to P_1 at event number 1; process P_{144} at event number 6 is sending a message to P_3 at event number 0; process P_{144} at event number 166 is sending a message to P_{83} at event number 0 and process P_{144} at event number 422 is receiving from P_{134} at event number 48. This

communication is displayed in blue as these processes were selected by the pickup filter while the unfiltered processes and their communication are grayed out.

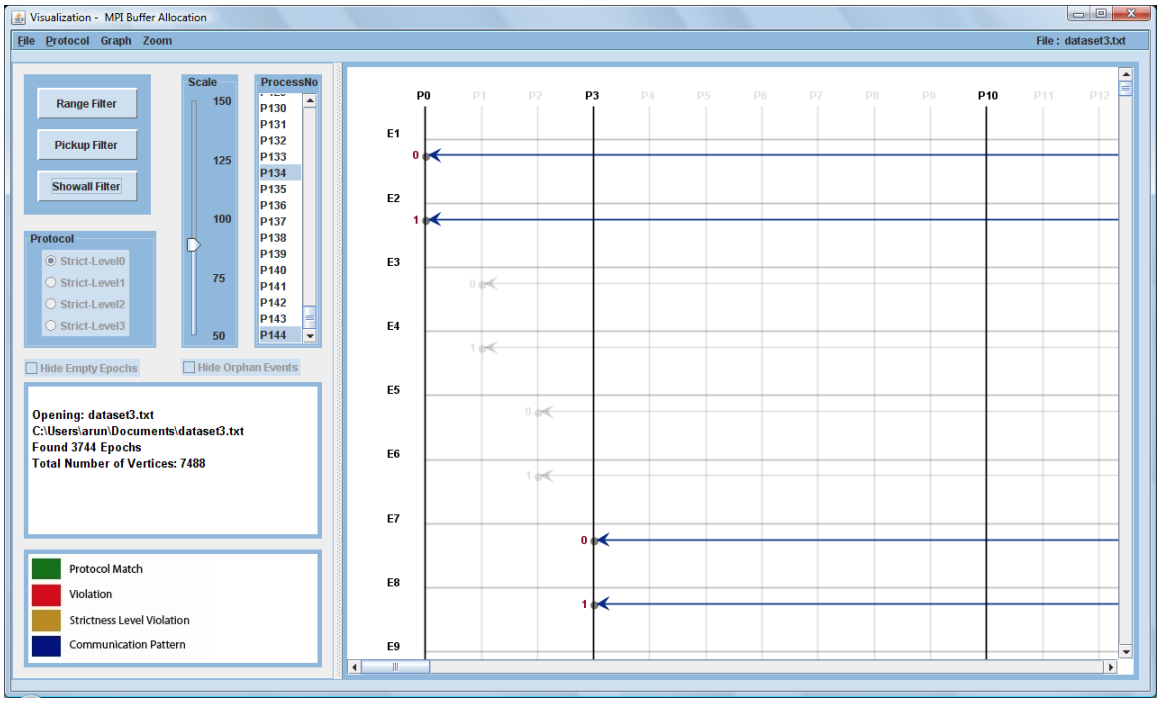

Figure 4-6a: Result of applying the showall filter.

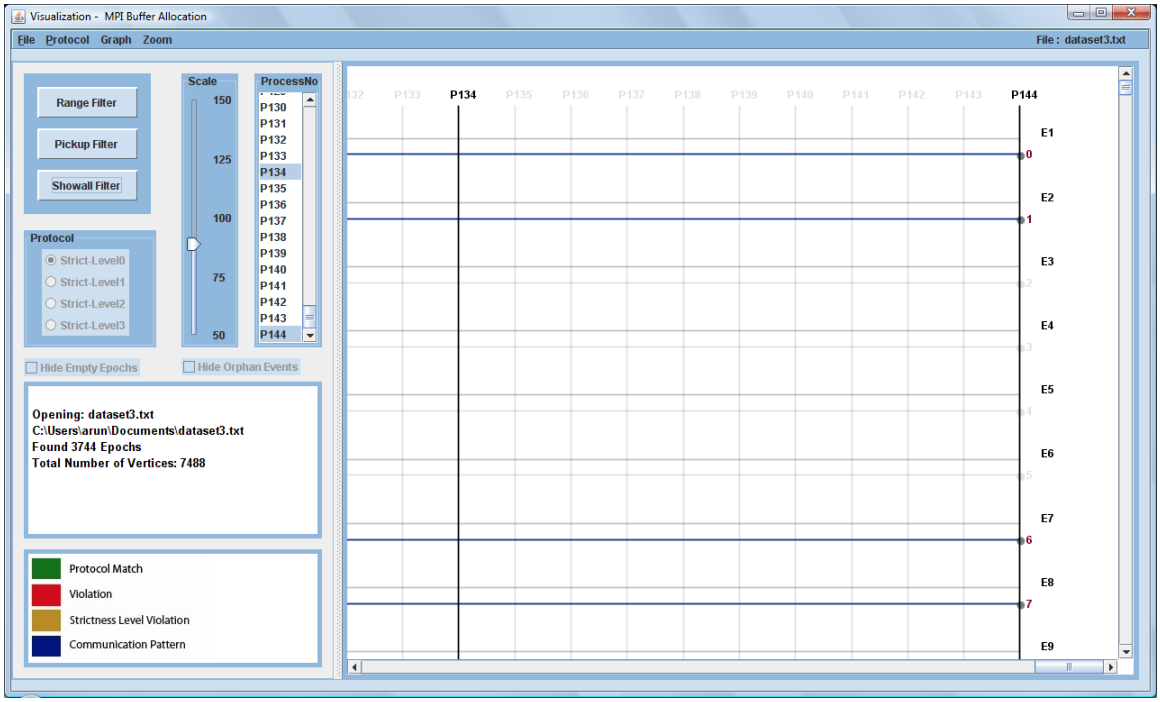

Figure 4-6b: Result of applying the showall filter.

Hide orphan Events

On checking the hide orphan events checkbox, the orphan events in the graph are hidden. Hence, the graph would contain only complete events and empty epochs. An event send (receive) is complete if its corresponding receive (send) node is also in the graph. In Fig 4-7a events 2, 3, 4 and 5 for P_{144} are the orphan events. On checking the hide orphan events checkbox, the orphan events disappear as displayed in Fig 4-7b.

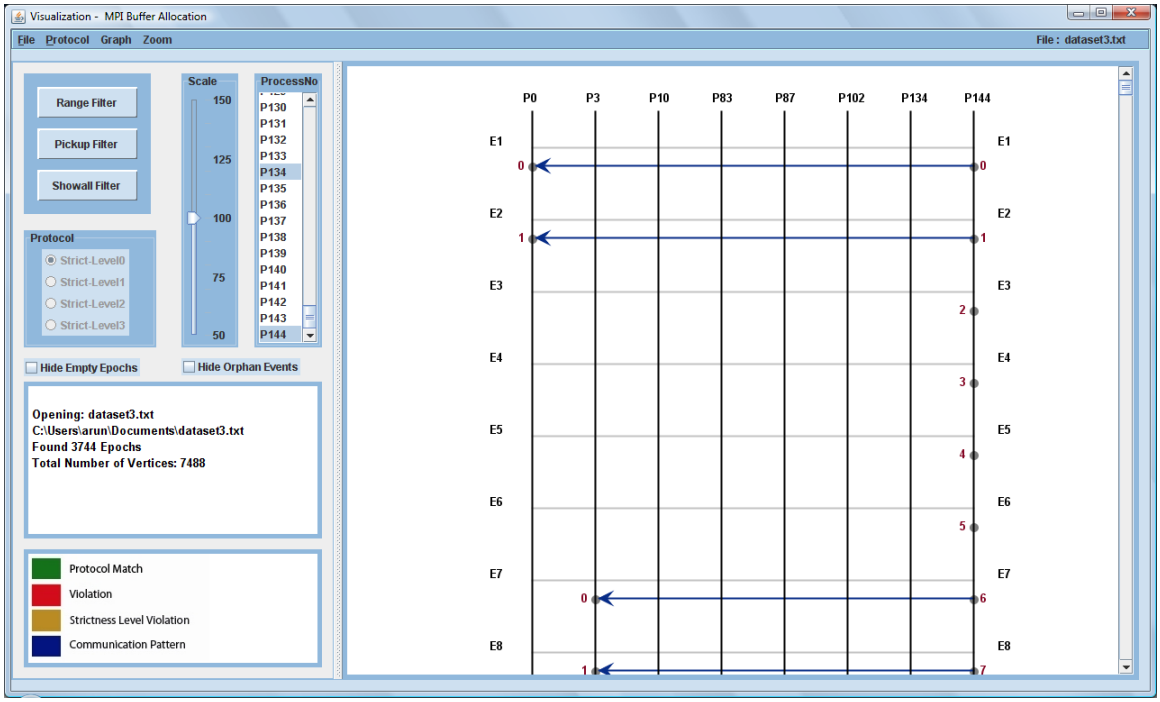

Figure 4-7a: Before hiding the orphan events.

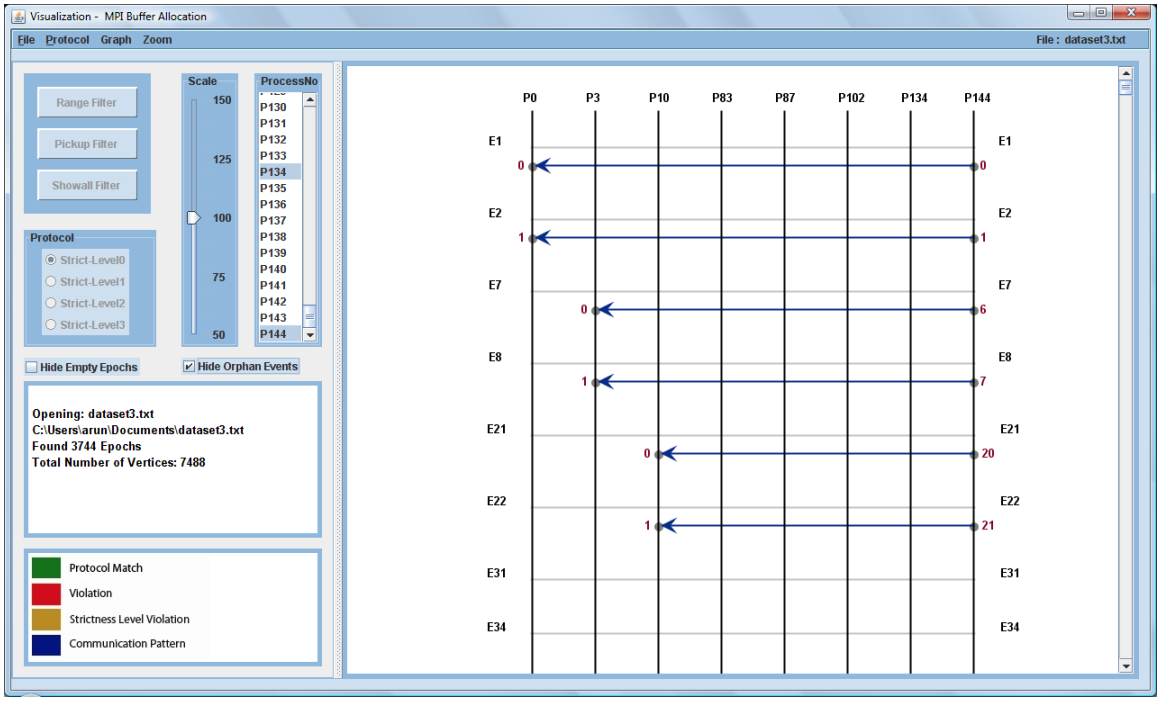

Figure 4-7b: After hiding the orphan events.

Hide Empty Epoch

On checking the hide empty epoch checkbox, all empty epochs in the graph are removed, thus leaving the graph with complete events and orphan events. Consider the graph in Figure 4-8a with empty epochs E_{31}, E_{34} and E_{37}. Figure 4-8b, shows the resulting graph on hiding the empty epochs.

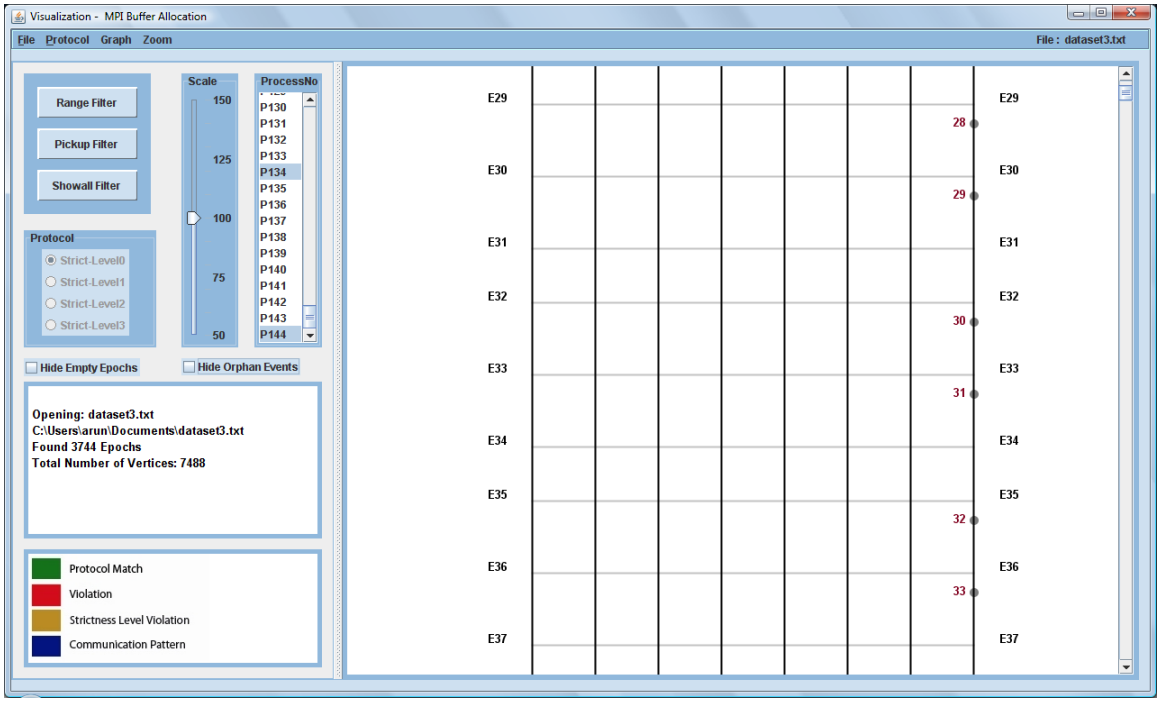

Figure 4-8a: Before hiding empty epoch.

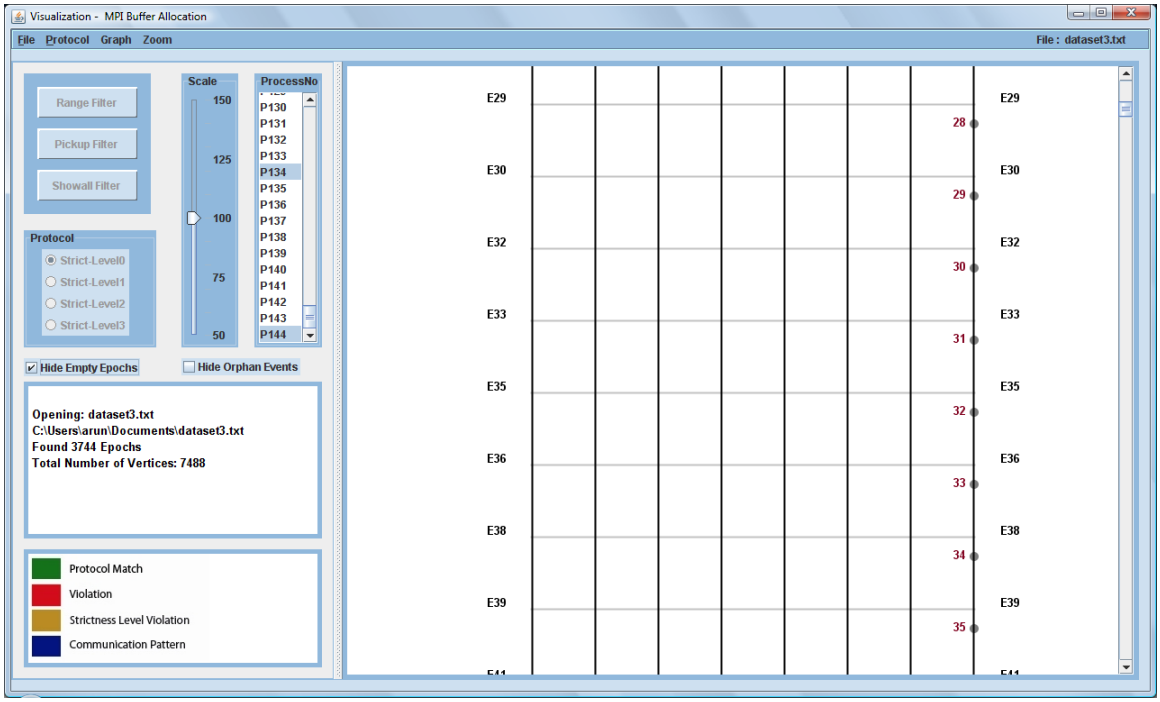

Figure 4-8b: After hiding empty epoch.

Hide both Orphan Events and Empty Epoch

On checking both (hide orphan events and hide empty epochs) checkboxes, empty epochs and orphan events are removed leaving the graph with only complete events as shown in Figure 4-9a and 4-9b. The hide orphan events or hide empty epochs checkboxes can be checked if and only if the graph was filtered by applying either range filter or pickup filter.

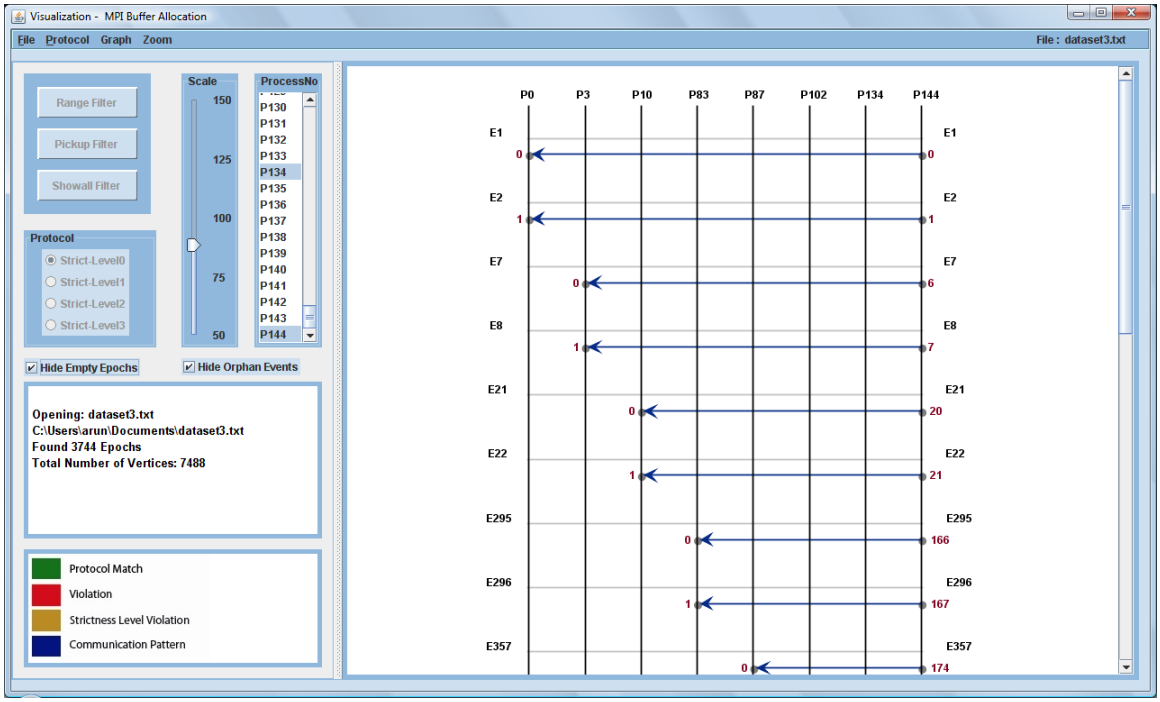

Figure 4-9a: Graph after hiding orphan events and empty epochs.

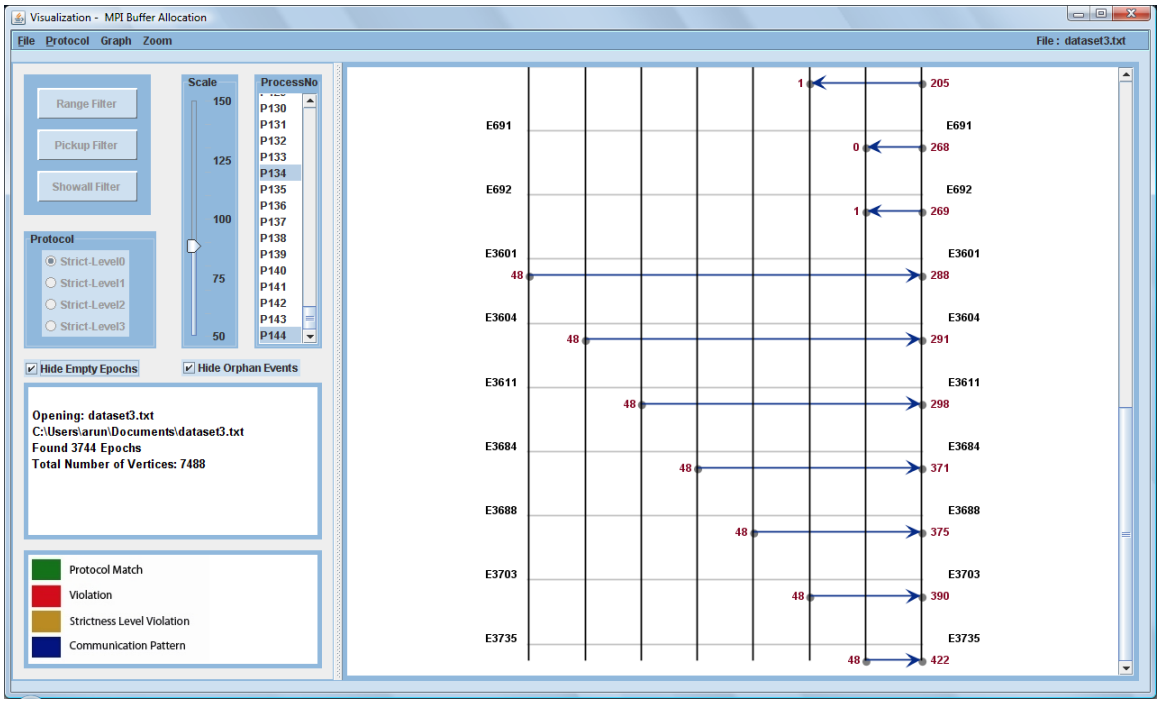

Figure 4-9b: Graph after hiding orphan events and empty epochs.

Protocol Conformance Checking and Strictness Level Violations

To illustrate the working of protocol conformance checking, we use the following dataset `DES-10-collect.txt` containing 10 process components. The corresponding graph is shown in Figure 4-10a and 4-10b.

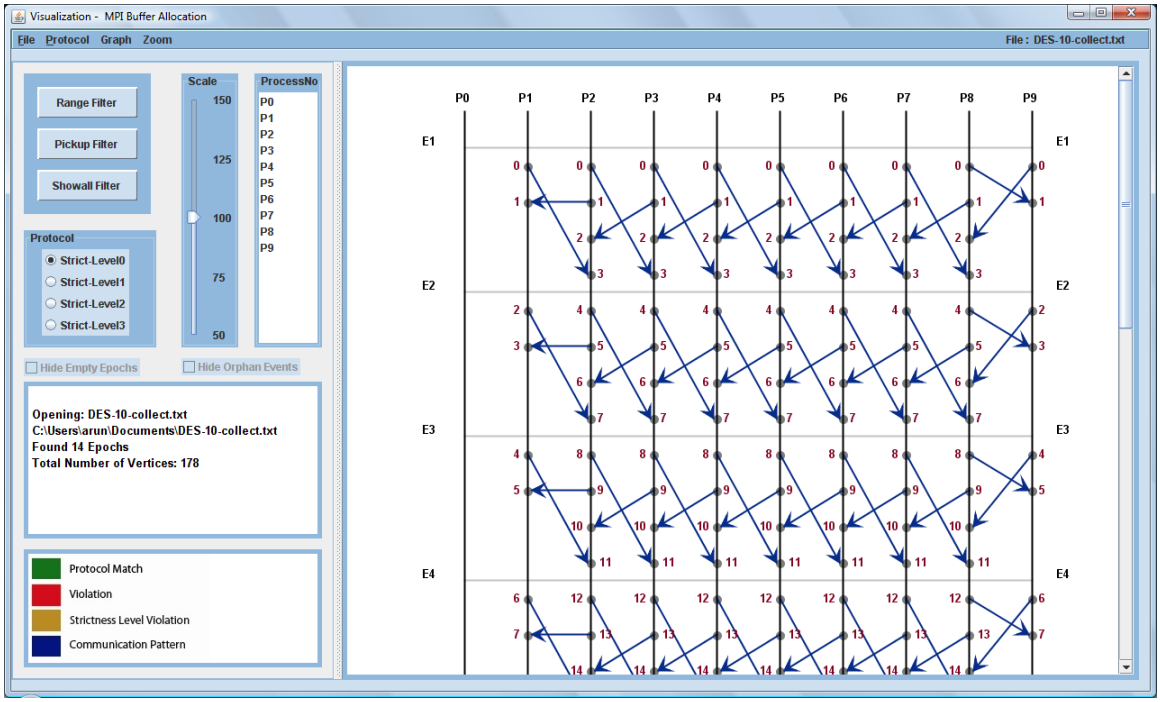

Figure 4-10a: Graph containing 10 processes (`DES-10-collect.txt`).

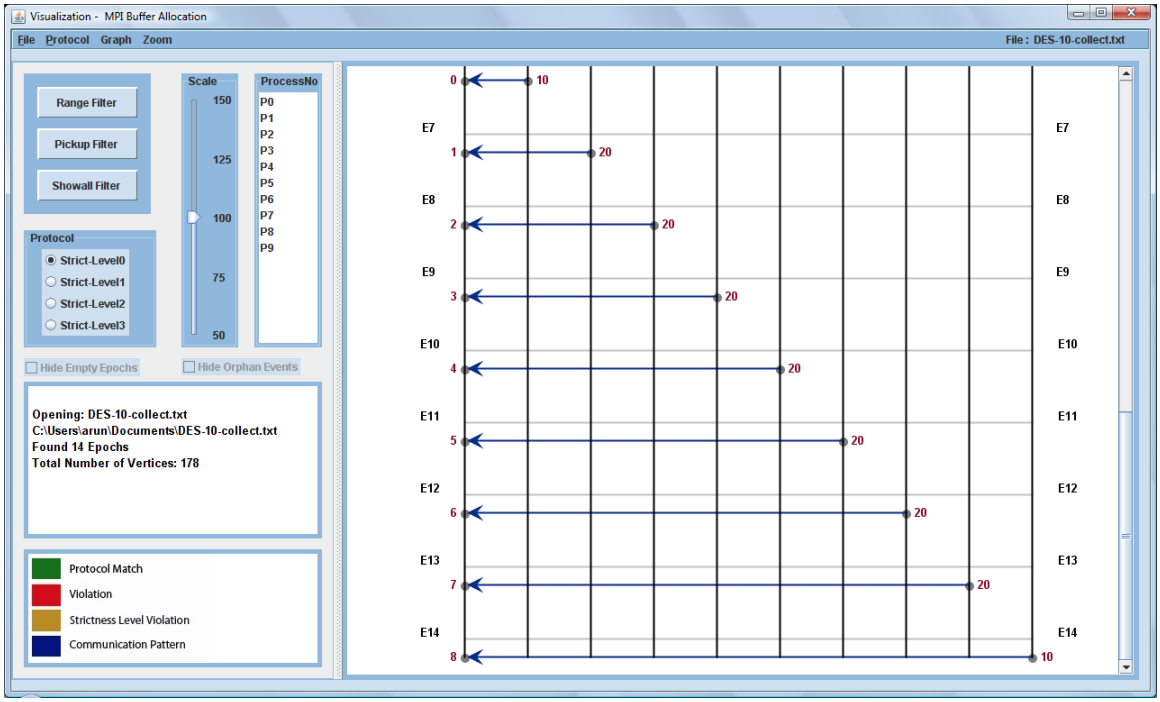

Figure 4-10b: Graph containing 10 processes (`DES-10-collect.txt`).

The protocol constraints are displayed in Figure 4-11. The first specification line in the Figure 4-11 specifies that any process at any line number can send a message to the process to its immediate right (less than or equal to P_8). To make the constraint a little more complex we introduce line numbers to the rest of the constraints. The second constraint specifies that any process except the master P_0 at line number 165 can send a message to any other process at line number 169. The third protocol constraint introduces an identifier n which denotes the total number of processes (n=10); it specifies that any process can send a message to its immediate right (less than or equal to P_8) except P_9 which can send only to the master P_0 (190 and 90 being the corresponding line numbers for the send and receive processes). Finally the last protocol constraint specifies that process P_9 at line number 190 can send a message only to master P_0 at line number 90.

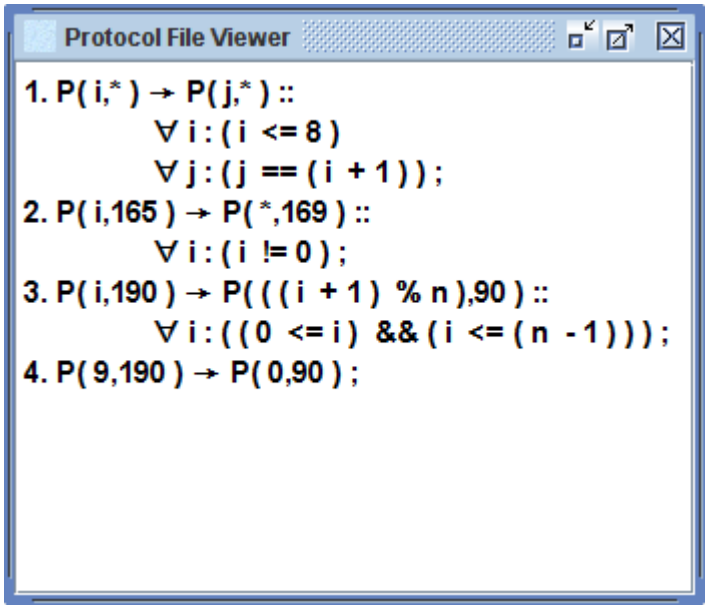

Figure 4-11: A protocol specification.

Upon applying the protocol specification to the graph in 4-10a and 4-10b, it is redrawn as shown in Figure 4-12a and 4-12b. Strictness level 0 is defined as follows: if a message violates a protocol line `i`, it must match atleast one protocol line to avoid violating the protocol. Since strictness level 0 is the default strictness level, it is applied to the graph when the protocol specification file is initially loaded and applied. All the messages that match the specification are colored green with violations colored in red. A protocol result for strictness level 0 is shown in Figure 4-13.

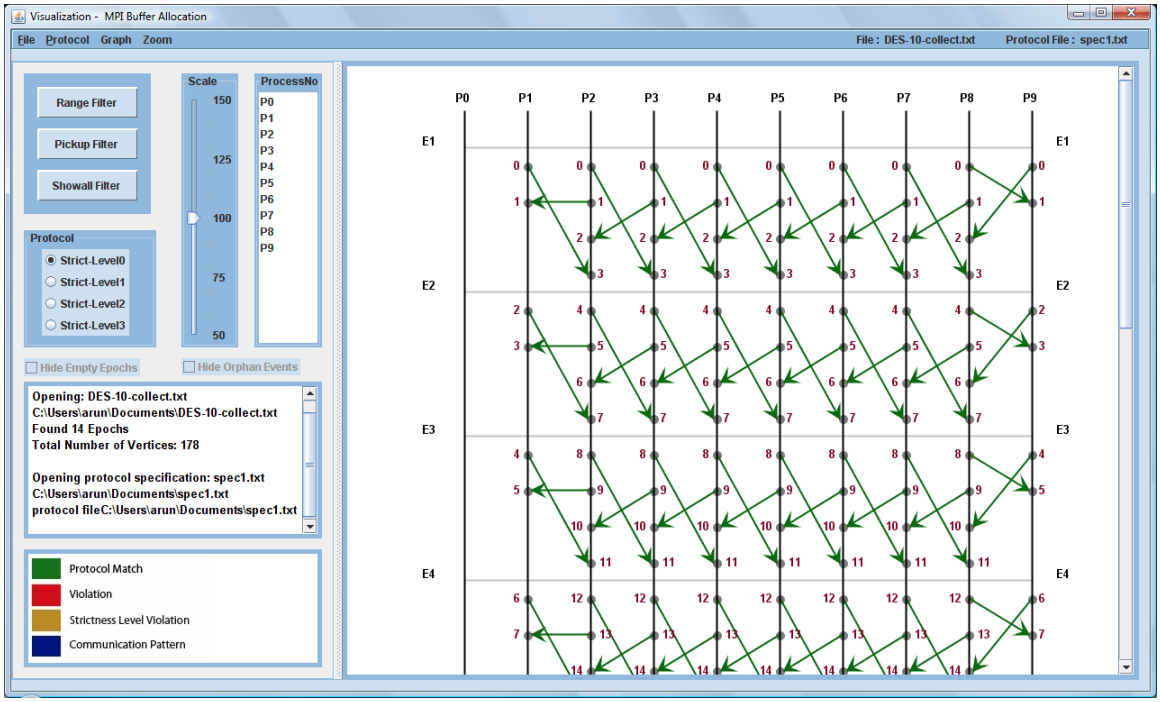

Figure 4-12a: After applying strictness level 0.

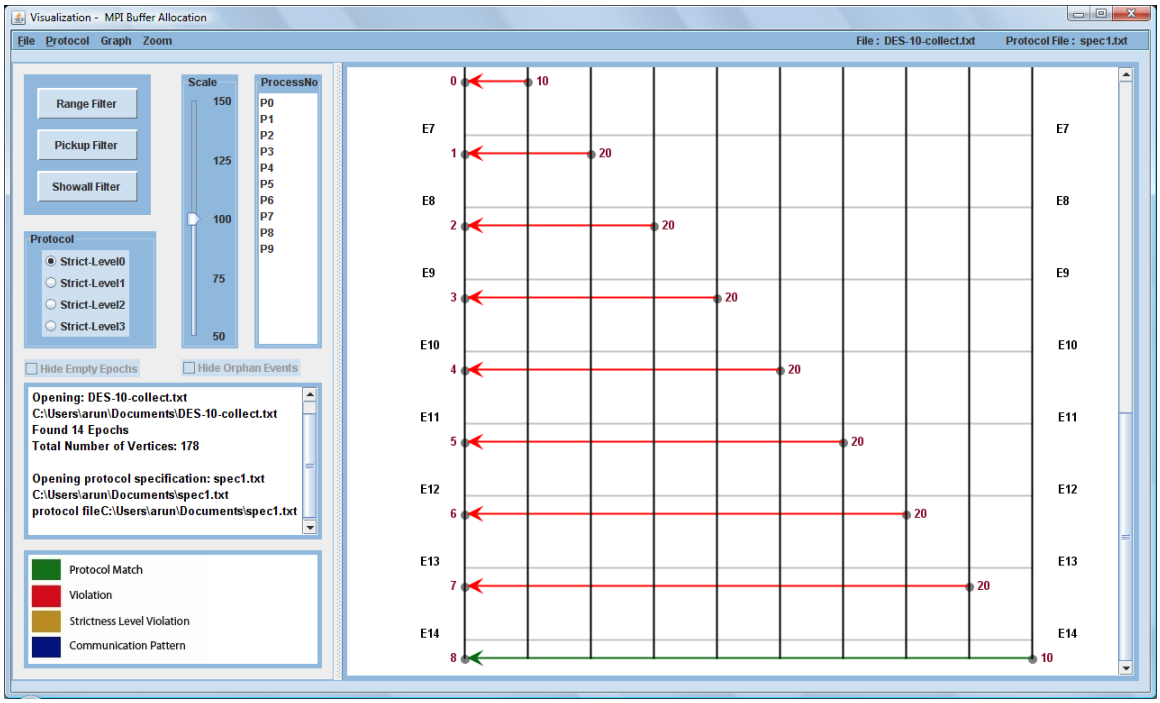

Figure 4-12b: After applying strictness level 0.

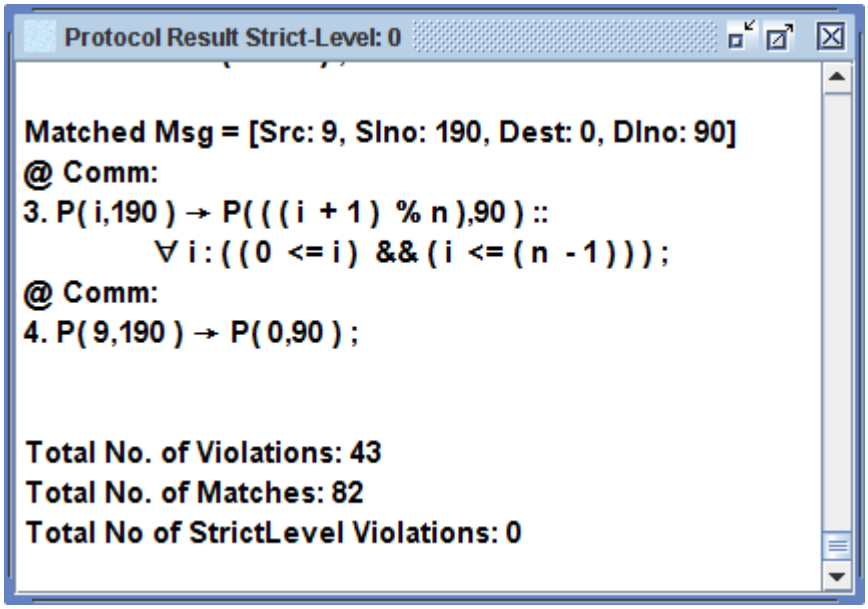

Figure 4-13: Protocol results for strictness level 0.

Strictness level 1 is defined as follows: a message can match 0 or more protocol lines `i` with 0 violations to avoid violating the protocol. On applying strictness level of 1, some messages which are colored green in Figure 4-12a are changed to orange and messages that are colored red in Figure 4-12b are changed to orange. All messages that are changed to orange color indicate that they have violated strictness level of 1. The resulting graph after applying strictness level 1 is shown in Figure 4-14a and 4-14b. A protocol result for strictness level 1 is shown in Figure 4-15.

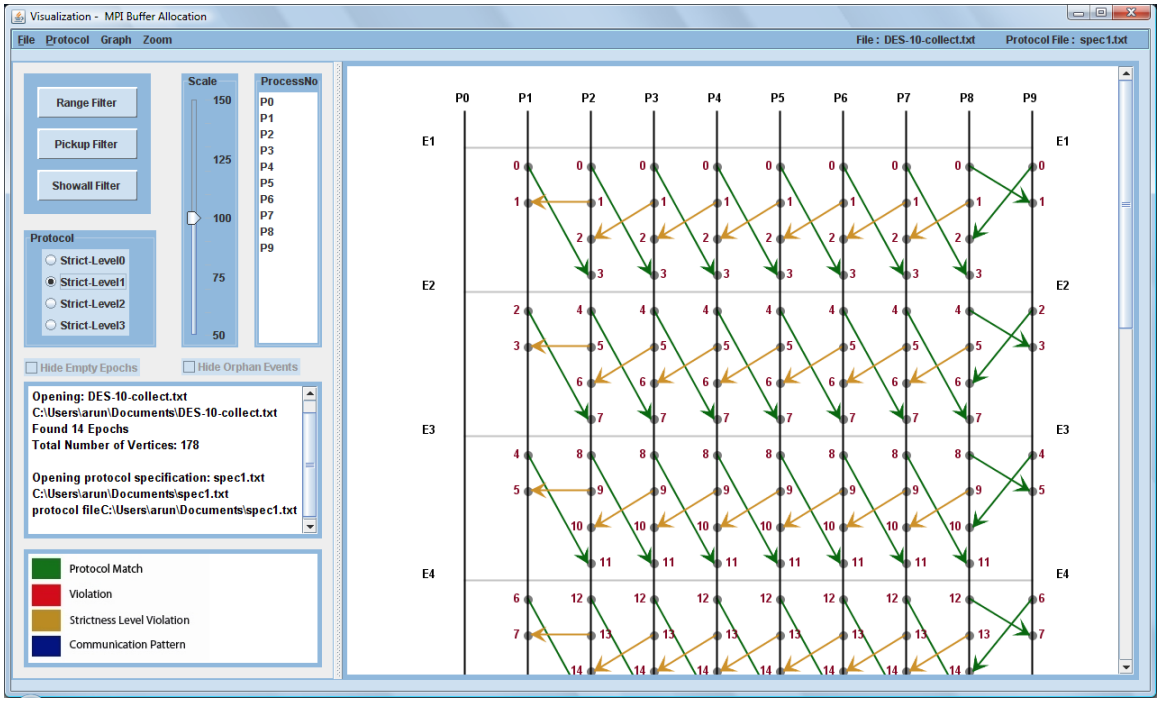

Figure 4-14a: After applying strictness level 1.

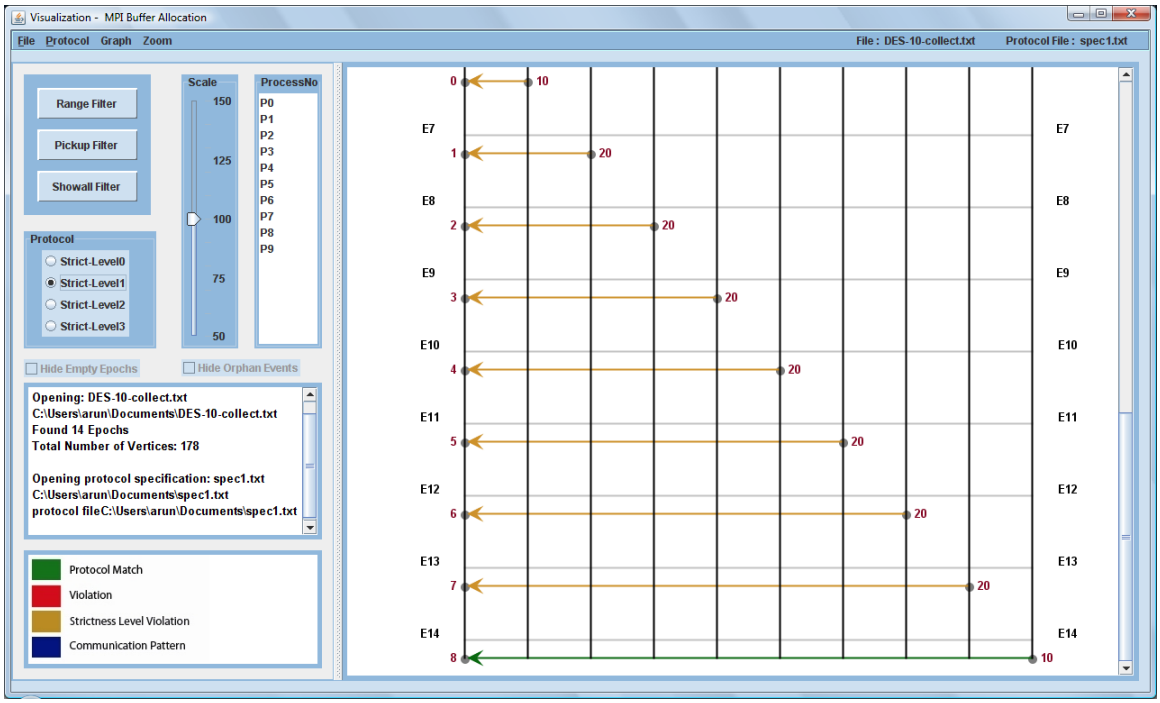

Figure 4-14b: After applying strictness level 1.

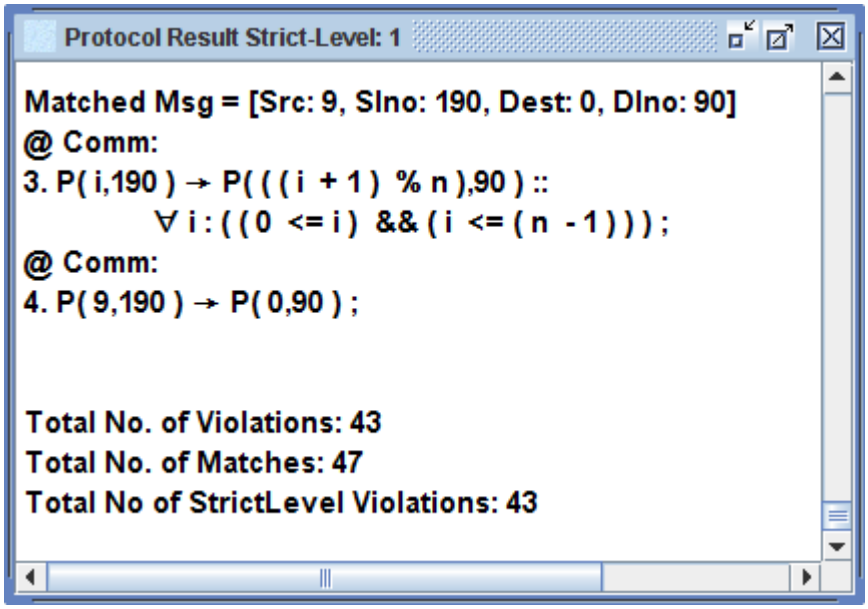

Figure 4-15: Protocol results for strictness level 1.

Similarly, when strictness level 2 is applied, all messages that violates the strictness level in Figure 4-12a and Figure 4-12b are changed to orange. The resulting graph, after applying strictness level 2 is shown in Figure 4-16a and Figure 4-16b. Strictness level 2 is defined as follows: a message must match atleast one protocol lines `i` with 0 violations to avoid violating the protocol. A protocol result for strictness level 2 is shown in Figure 4-17.

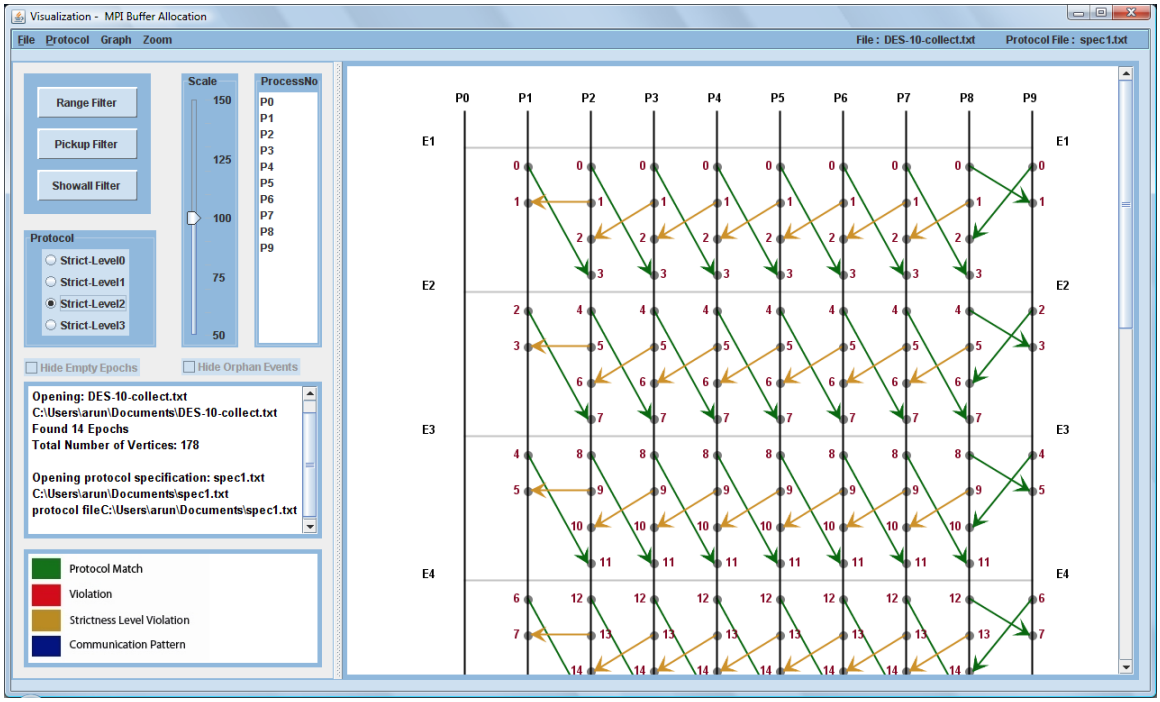

Figure 4-16a: After applying strictness level 2.

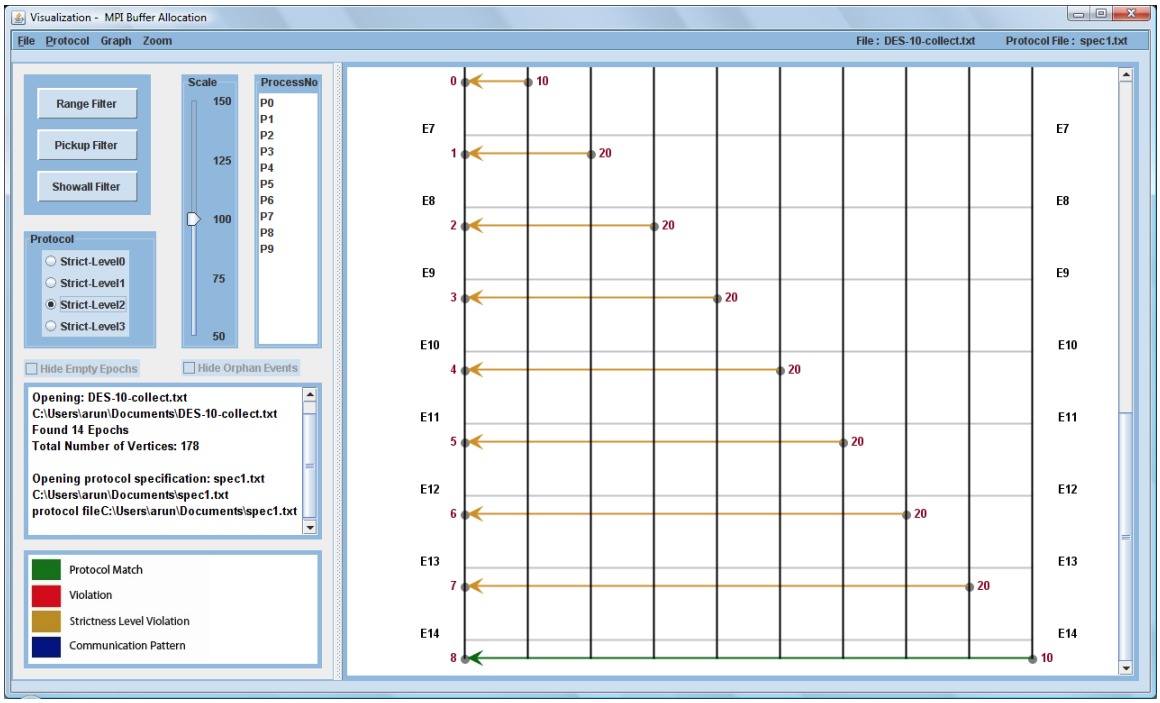

Figure 4-16b: After applying strictness level 2.

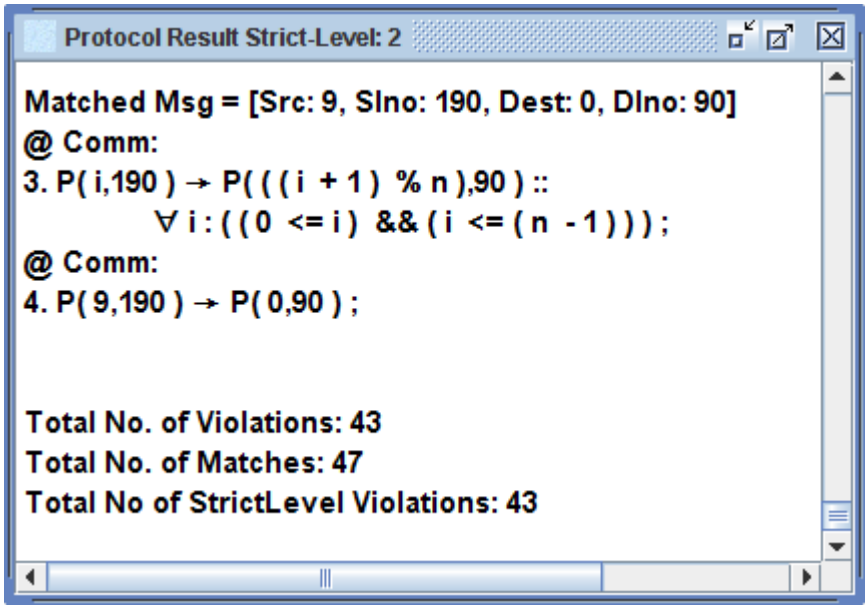

Figure 4-17: Protocol results for strictness level 2.

Strictness level 3 is defined as follows: a message must match exactly only one protocol line `i` with 0 violations to avoid violating the protocol. Strictness level 3 really requires a "fully specified" protocol constraint specification. Figure 4-18a and 4-18b are the graphs obtained on applying strictness level of 3. The graph is similar to the previous resulting graph obtained after applying strictness level 1 or strictness level 2. Furthermore, the message from P_9 at event number 10 to P_0 at event number 8 with epoch number E_{14} is colored orange as it violates strictness level of 3. A protocol result for strictness level 3 is shown in Figure 4-19.

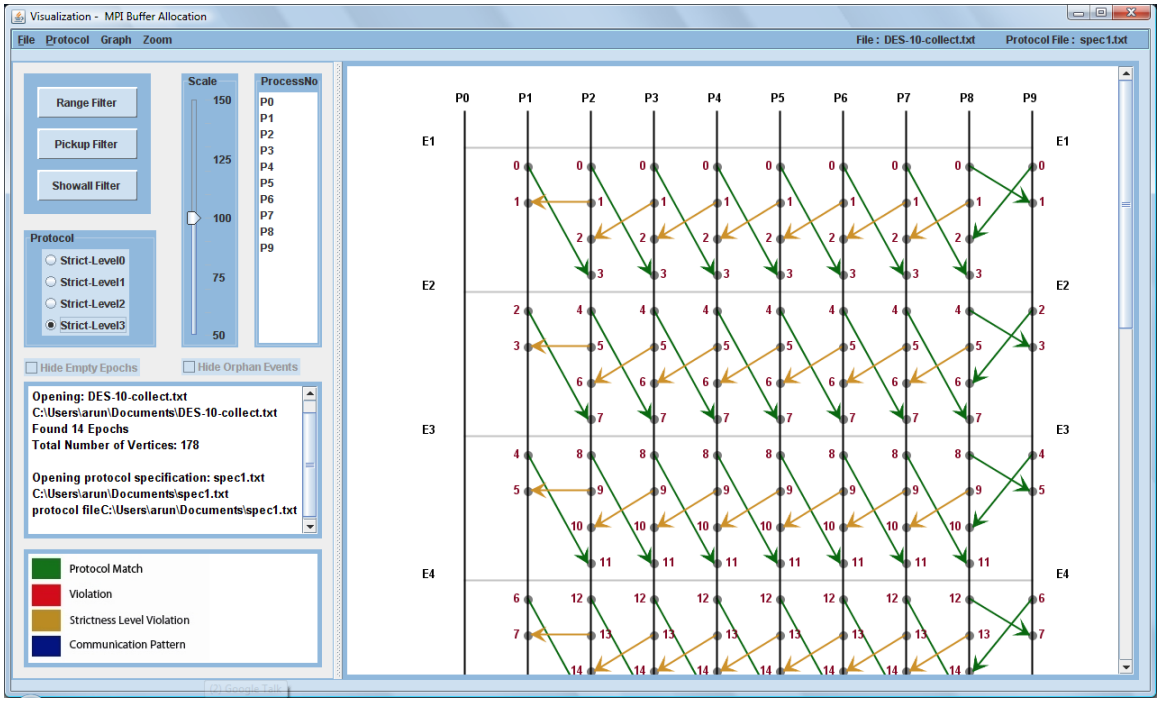

Figure 4-18a: After applying strictness level 3.

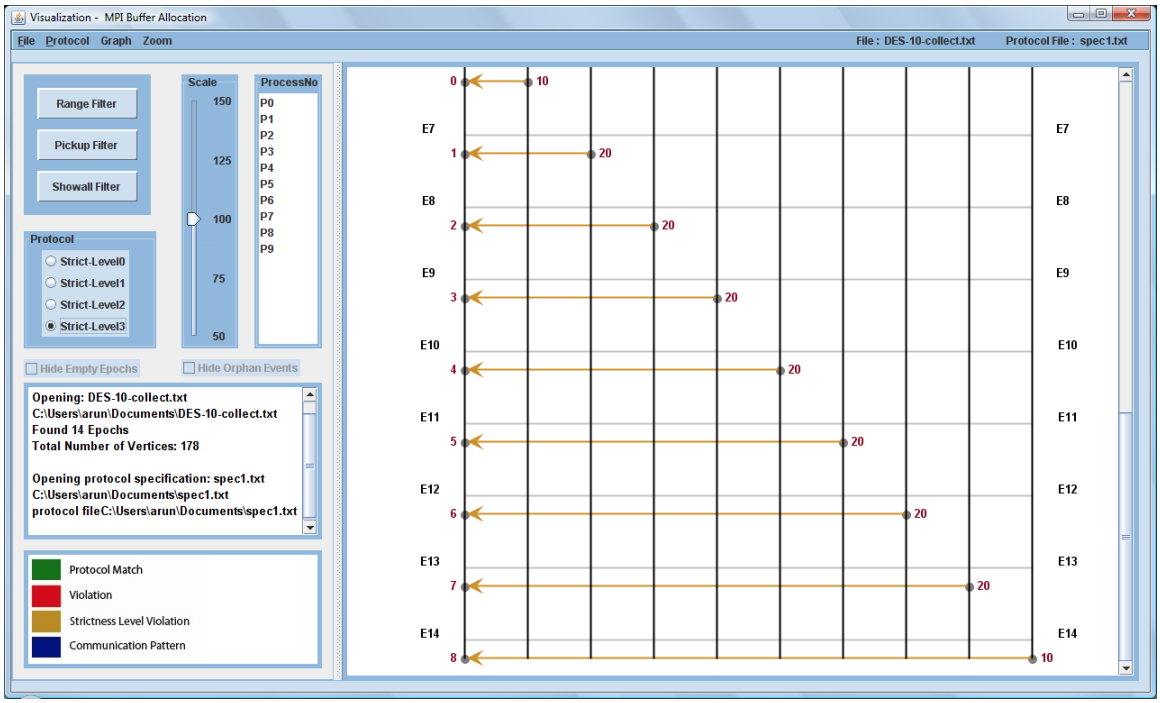

Figure 4-18b: After applying strictness level 3.

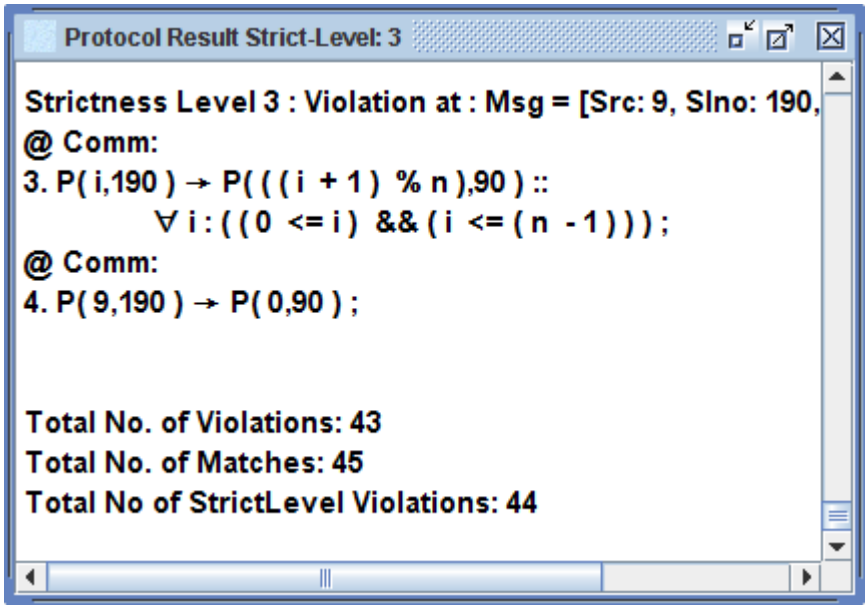

Figure 4-19: Protocol results for strictness level 3.

Node Level Tool Tip

This feature is used to display information about the send nodes and send or receive nodes of orphan events. Upon clicking the node of process P_5 at event number 0, the node level information such as its destination, source, epoch number, event numbers along with protocol results are displayed. Since the message matches a protocol line, the corresponding protocol constraint is also displayed. This is illustrated in Figure 4-20.

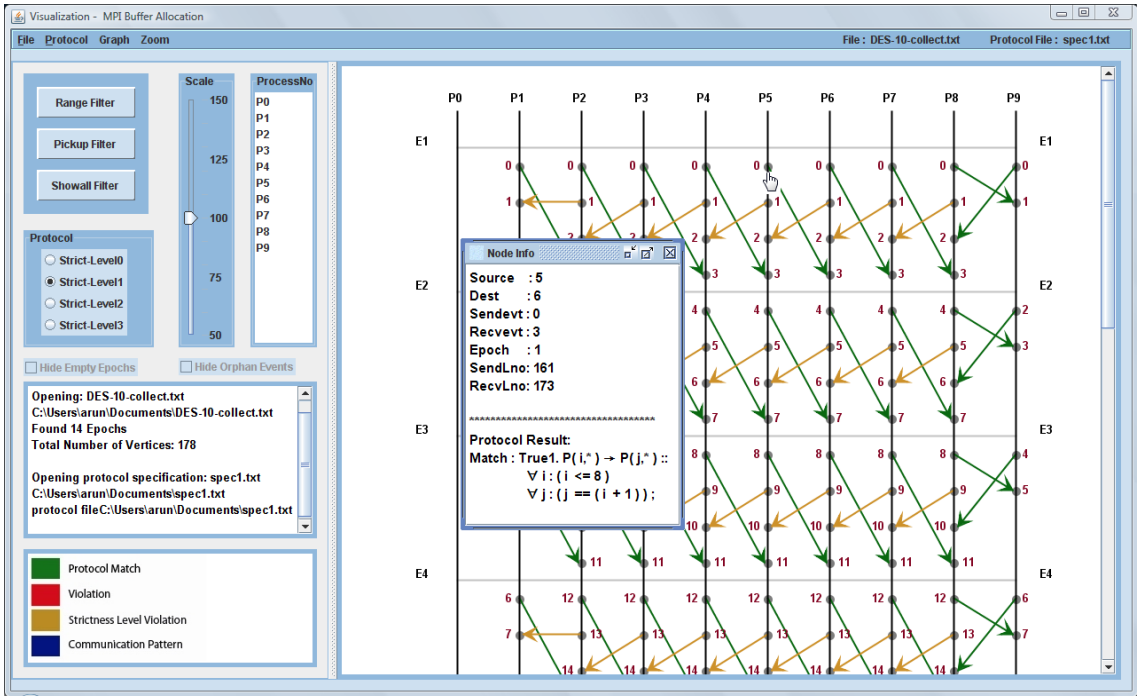

Figure 4-20: Node info.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we developed a graph visualization tool that visualizes log data from message passing systems. The graph structure is similar to Lamport space-time diagram [5] containing a set of process components and communication taking place between the process components. We also implemented a protocol conformance checking algorithm which enables the user to write a protocol specification file containing a list of constraints that the messages should satisfy. The tool includes interaction techniques such as spot zoom, node level tip and filter options to reduce the amount of information displayed on the screen and also to navigate through the graph in an effective manner especially when the graph is large. The tool was effectively tested against huge log files containing as much as 145 processes with 3,744 epochs and 7,488 vertices. The messages in the graph were checked against simple protocol constraints and finally with a complex one that contains line numbers. These messages were also tested at different strictness levels to ensure the correctness of the protocol conformance checking.

Future Work

The following list contains possible feature enhancements that could be made in future.

- The tool presently is designed to visualize only one dataset at a time. This may be extended to support visualization of multiple datasets concurrently.

- Advanced filtering may be supported enabling the user to filter not only by processes but also by epoch numbers, event numbers or even protocol conformance results.
- The zoom feature cannot be used to zoom into specific areas of the graph. A zoom and pan feature may be more effective.
- The graph structure currently supports epochs that separate the execution interval; the tool may be enhanced to include super epochs. A super epoch is a consecutive sequence of one more epochs. The use of super epochs in the approximation of the buffer allocation problem is to reduce the overhead associated with the barrier synchronization needed after each epoch.
- Currently the graph is drawn based on fixed log file or dataset file, thus performing an offline analysis. A real time log file evaluation drawing dynamic graphs would be an interesting enhancement.

REFERENCES

[1] http://www.infovis.org/. (Last visited Feb/ 2010).

[2] http://people.cs.vt.edu/~tripathi/AlgoViz/. (Last visited Feb/ 2010).

[3] A. Brodsky, J.B. Pedersen, and A. Wagner. *On the complexity of buffer allocation in message passing systems*. Journal of Parallel and Distributed Computing, vol 65, No 5 (June) 2005, pages 692-713.

[4] J.B. Pedersen, A. Brodsky, J. Sampson. *Approximating the Buffer Allocation Problem using Epochs*. Under the review for the Journal of Parallel and Distributed Computing, vol 68, No 9 (September) 2008, pages 1263-1282.

[5] L. Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*. Communications of the ACM July 1978 Volume 21 Number 7.

[6] I. Herman, G. Melançon, and M.S. Marshall. *Graph Visualization and Navigation in Information Visualization: a Survey*. Visualization and Computer Graphics, IEEE Transactions on Volume 6, Issue 1, Jan-Mar 2000 pages: 24-43.

[7] J.B. Pedersen. Multilevel *Debugging of Parallel Message passing programs*: Ph.D. Thesis, University of British Columbia, 2003.

[8] D. Kimelman, B. Leban, T. Roth, and D. Zernik. *Reduction of Visual Complexity in Dynamic Graphs*, Proceedings of the Symposium on Graph Drawing GD ’93, Springer–Verlag, 1994.

[9] D.A. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, and B.W. Schwartz, *An overview of the Pablo performance analysis environment*, technical Report, Dept. of Computer Science, University of Illinois, Urbana-Champaign, 1992.

[10] N. Stankovic, K. Zhang. *Graphical Composition and Visualization of Message Passing Programs*. Proceedings of Software Visualization Workshop, 1997.

[11] B.B. Bederson and B. Shneiderman (2003). *The Craft of Information Visualization: Readings and Reflections*, Morgan Kaufmann ISBN 1-55860-915-6.

[12] W.E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, K. Solchenbach. V*AMPIR: Visualization and Analysis of MPI Resources*. Proceedings of the 21st International Supercomputer Conference, June 2006.

[13] http://www.caida.org/tools/visualization/walrus/. (Last visited Feb/ 2010).

[14] K. Fürlinger and D. Skinner. *Capturing and Visualizing Event Flow Graphs of MPI Applications*. In Workshop on Productivity and Performance (PROPER 2009) in conjunction with Euro-Par 2009, August 2009.

[15] B. Schaeli, Ali Al-Shabibi and R.D.Hersch. *Visual Debugging of MPI Applications*. Proceedings of 15th European PVM/MPI User'Group Meeting (EuroPVM/MPI), September 2008.

[16] J. Bruck, D. Dolev, C. Ho, M. Rosu, and R. Strong. *Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations*. Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 64-73, July 1995.

[17] K. Kaugars, R. Zanny and E.de Doncker. *PARVIS: Visualizing Distributed Dynamic Partitioning Algorithms*. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications.

[18] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V.S. Sunderam. *PVM 3 User's Guide and Reference Manual*, Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993.

[19] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[20] Formal Systems (Europe) Ltd., *Failure-Divergence Refinement*. FDR2 User Manual, 14 June 2005.

[21] J. Pedersen, A. Wagner. *PVM-Builder - A Tool for Parallel programming*. Proceedings of the 5th International Euro-Par Conference, Toulouse, France, pages 107-112, August/September 1999.

VITA

Graduate College
University of Nevada, Las Vegas

Arunkumar Sadasivan

Degrees:

Bachelor of Engineering, Computer Science, 2005
Anna University, India.

Thesis Title:

A Visualization Approach for Message Passing in Parallel Computing System

Dissertation Examination Committee:

Chairperson, Dr. Jan B. Pedersen, Ph. D.
Committee Member, Dr. Kazeem Taghva, Ph. D.
Committee Member, Dr. Yoohwan Kim, Ph. D.
Graduate College Representative, Dr. Henry Selvaraj, Ph. D.