Los Alamos National Laboratory I/O Visualization Tool

Final Design Report

Chaii Lee & Danny Morris

June 18, 2008

Abstract

Los Alamos National Laboratory is a leading national security research institution that provides scientific and engineering solutions for the nation's most complex problems. Their focus is primarily on safety, security, and reliability of our nation's defenses. Los Alamos National Laboratory also works to advance the fields of computer science, bioscience, material science, chemistry, and physics.

In order to support these large research efforts, LANL has developed some of the world's most powerful supercomputing clusters. Their performance and reliability is vital to supporting their efforts. This project aims to provide a visualization solution to help analyze the I/O traces of those large clusters. Since much of the data is classified, this visualization solution provides a method for individuals outside of LANL to analyze the patterns and performance as well as simulate those computing clusters.

Table of Contents

Executive Summary

The goal of this project was to enhance our client's visualization software utilized by a group at the Los Alamos National Laboratories. The group conducts analysis and research in storage and I/O patterns of very large computing clusters. Much of the data used is classified so their research is unseen by the public. The visualization software that we were required to enhance allows for the analysis of these very large systems without displaying vital information while providing a simulation of the data.

Our client required us to make three enhancements to the software that he felt were most essential to improving its usability. The software needed to be able to wrap the information on the screen so there was little horizontal scrolling, the size of the output needed to be adjustable so the user could see relevant information, and the coloring scheme altered so it was more uniform and visually appealing. Over the course of the project we altered some of our original plans for the requirements but were able to add additional features and provided a framework that could be built on and expanded in future implementations.

We accomplished our requirements using Perl/Tk; for that was the language the software was written in. We were very pleased with the results of our project. We were able to deliver the desired features as well as add some additional features that we felt could benefit the future development of the software. We were also able to gain more experience using Perl along with the graphical interface extension Tk, which was fairly easy to pick up after examining the existing code.

Introduction

Los Alamos National Laboratory is a leading national security research institution that maintains many of the world's supercomputers in addition to many other things. LANL's effort to maintain current systems as well as the design of future systems requires an in-depth knowledge of how these supercomputers work. In doing so, the Storage and I/O group at LANL has developed and maintained a trace mechanism to capture the I/O usages of these supercomputers.

A large community of individuals whose interests are in these I/O traces has led to a high volume of demand for these I/O traces. Unfortunately, because of the classified nature these I/O traces store much, if not all, of these I/O traces cannot be released to those individuals. In response, LANL has requests for the development of an additional mechanism to turn their classified I/O traces into useful information for those non-LANL individuals.

Project Overview

Our client John Bent has provided a version of Los Alamos National Laboratory's visualization tool used to simulate I/O traces. Although the visualization tool works to some degree, the visualization tool itself is still in the development phase and lacks various features. For this project, our goal is to implement additional features at our client's request as well as refine current features. In addition, another goal for this project involves simulating unclassified versions of LANL's I/O traces to verify the accuracy of the visualization tool.

Project Requirements and Specifications

Given the scope of the project, we divide the project up into functional and nonfunctional requirements and specifications for a better understanding.

Functional

To reach our goals, some functional requirements and specifications we must meet include:

- Implementing a feature that will allow the visualization tool to adjust, redraw, and wrap around files when they reach the end of the screen
- Implementing a feature that will allow the user to set which will redraw individual files to various size
- Implementing a feature that will uniformly assigns a color value from the color spectrum to individual I/O processes

As stated by our client, it is preferred that the first two options be implemented since they are features needed for functionality. Should we complete the first two options and have additional time, we can implement the third option for it is only an aesthetic feature. Our client was very flexible with the

requirements provided that the first two features get implemented in some fashion. We were encouraged to explore additional features and make any new changes if time permitted.

Nonfunctional

As for nonfunctional requirements and specifications, they include:

- Using Perl/Tk for our implementations (the programming language LANL's visualization tool is written in)
- Have the tool use LANL's I/O trace files
- Accommodate the following I/O patterns: N-1 Non-Strided, N-1 Strided, and N-N

In order to gain knowledge in Perl/Tk, we will trace through existing code to see how the current version is implemented as well as use internet resources to find additional information when implementing our requirements. As for how the visualization tool works, it takes as input the trace files containing different I/Os for a particular job. The visualization tool reads these trace files line by line and selects certain events. Depending on the event, the software either draws a read event or a write event to the screen. Additional information on how these trace files are formatted and the types of I/O patterns used are available at http://institutes.lanl.gov/data/tdata/.

Project Design

To achieve our project requirements and specifications, we divide the project into three different components. The first component deals with how to wrap files around once they reach the end of the screen. The second component manages how to get and set a user-defined argument for outputting files. Finally, the third component deals with how to uniformly assign colors to each process.

Wrapping Large Files

The current program visualizes one file per line. No matter what type of data set is in use or the size of the file, all of the reads and writes for each process displays on a single line. The visualization fits ideally on the window for some of the data sets, but not for others. On the N-1 data sets where there is only one file, the visualization goes beyond the window and the user must scroll a long way to see the visualization. This is not very useful since they cannot see all of the information in one window. The client would like the output to wrap around on the screen so that all of the information can be seen by just scrolling vertically.

The ideal output is where the files end right at the edge of the window, however this only occurs on certain data sets (see figure 1). On sets that are too large or when there is only one file, we want the output to wrap around onto the next line while shifting all files beneath down the window. This issue usually occurs when we run an N-1 dataset (see figure 2).

Figure 1: Ideal Output

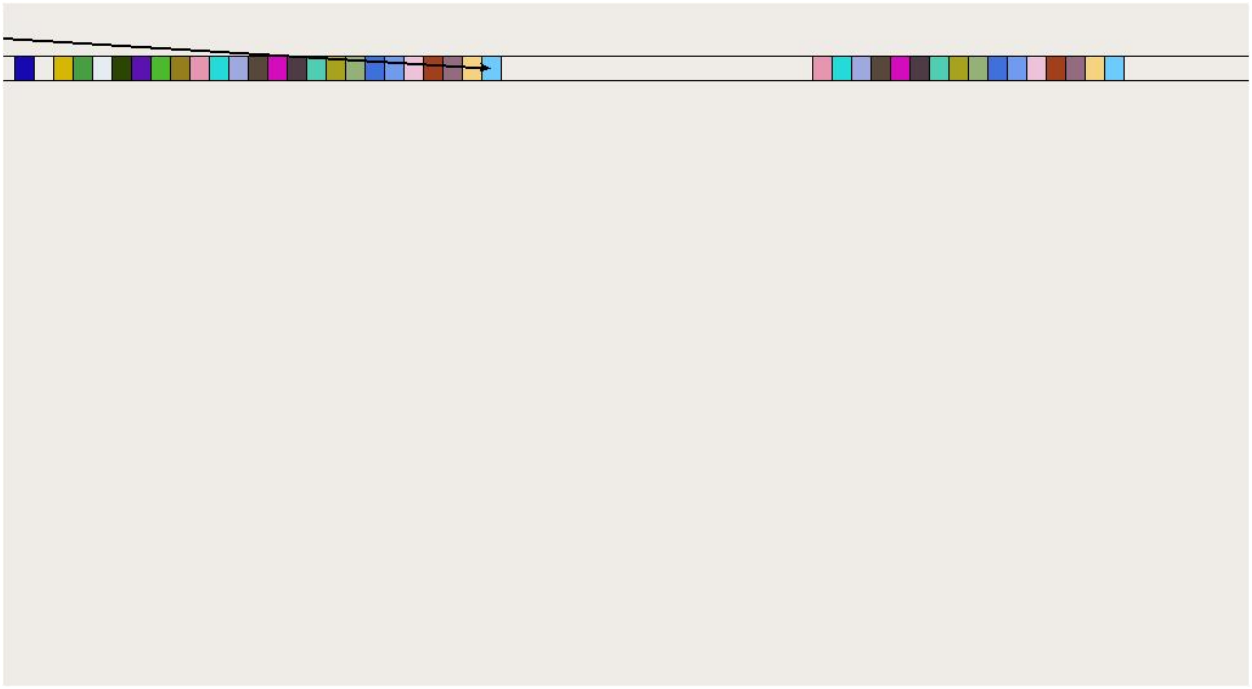

Figure 2: An N-1 Data Set with Only a Single Output File

The amount of wasted space is evident when the visualization tool is only writing to a single line. There is not any useful information for the user if they have to continually scroll to see where the file is being written to or read from.

Approach to Wrapping Large Files

To solve this issue we check to see if a particular file reaches the edge of the window. If it does then we shift all the files down the window and wrap the one that reaches the end onto the next line (see pseudocode 1). The tool does not currently store the positions of the items on the window. We plan on implementing a data structure that stores the locations of the output files in the window. If we should need to redraw the window when a file reaches the edge, we are able to access the current positions of all the files and shift them down so that they are not altering or overlapping with the file above it.

```
...
sub drawEvent {
      ...                    # previous code
      if ( file == width of screen )
            shiftScreen()
      else
            updateCanvas()
      ...                    # previous code
}
...
```

Pseudocode 1: Implementing Wrapping Large Files

Setting the Bytes-per-Pixel Flag

Currently, there are only two options to set and use the bytes-per-pixel flag (-bp flag). From the command line, if the user does not specify the –bp flag, then the visualization tool will automatically use the default argument. However, if the user specifies the –bp flag and an argument, then the visualization tool will overwrite the default argument and use the user-defined argument to set the bytes-per-pixel (see figure 3).

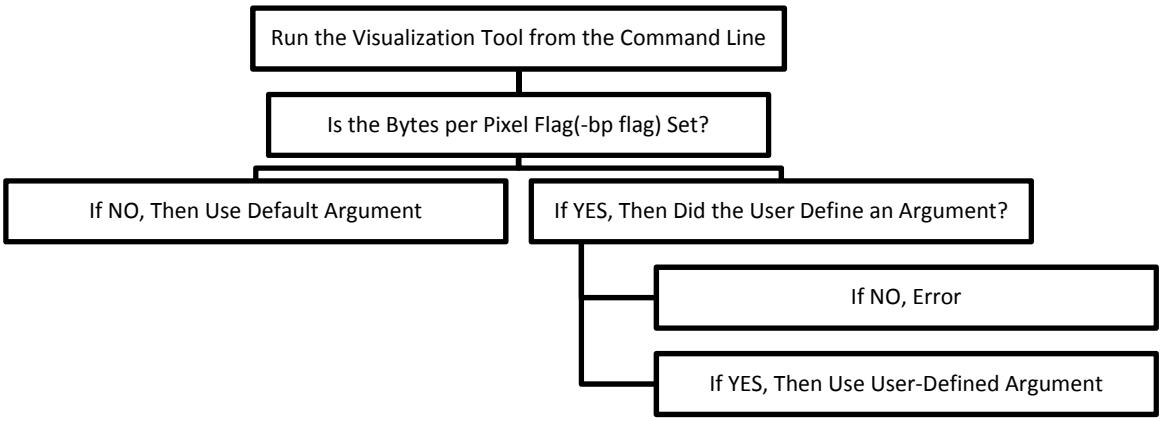

Figure 3: Flow Diagram for Current Bytes-per-Pixel Flag

The problem with this is that the user can only set the bytes-per-pixel once. In doing so, the user would have to use the same value throughout the visualization's runtime. This become problematic when the

default value or the value the user has specified does not work well with the I/O traces the visualization tool is simulating. Using the same I/O traces, if the user sets the –bp flag too small, the files become too large and the files are outputted beyond the screen. If the user sets the –bp flag too large, the files become too small and bunch up. In this example, using the default value produces a modest output (see figures 4-6).

Figure 4: Using a Small –bp Value

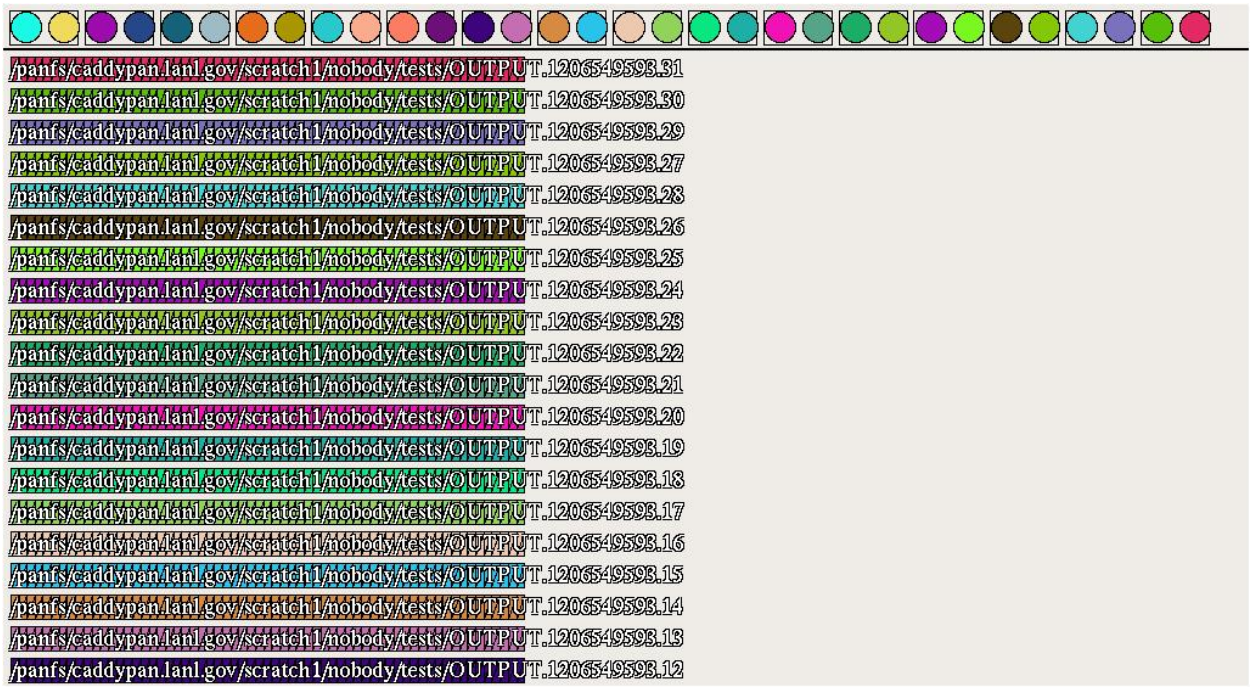

Figure 5: Using a Large –bp Value

Figure 6: Using the Default –bp Value

To solve this problem, the client has suggests that there be a method in which the users can right-click and manually adjust the –bp flag for each process. In doing so, the visualization tool automatically redraws the selected file with the new –bp value (see figure 7).

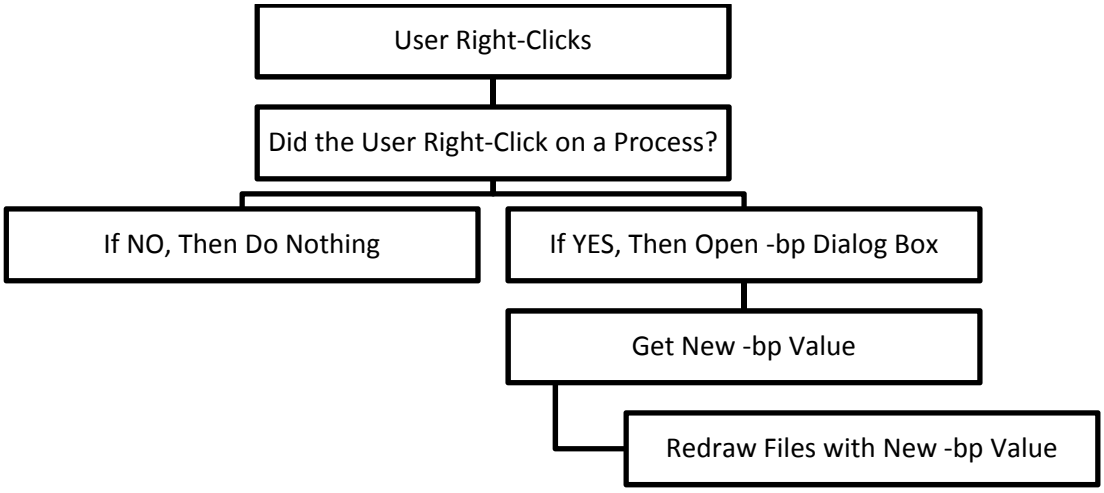

Figure 7: Flow Diagram for Bytes-per-Pixel Flag Design

Another problem with the current implementation is that there is only one global –bp value. This means that the default value or the value the user has specified has an effect on all the files. In other words, to adjust one file, this in return adjusts all the files to the new –bp value. A simple solution the client is suggesting is for each file to maintain a unique –bp value for the duration of the program. In doing so, the user can adjust one file without it affecting all the other files.

Approach to Setting the Bytes per Pixel Flag

Since each process maintains a data structure of values to uniquely identify each process, an addition of another value to the data structure keeps a unique –bp value. We can then initialize the –bp value for each process to the default value or to the value specified at the command line. When displaying each file to the screen, it can then use the –bp value for a particular process instead of the global –bp value (see Pseudocode 2).

```
...
struct File {
      ...                    # previous code
      bp_value => '$'        # new value to hold -bp value
      ...                    # previous code
};
...
sub drawEvent {
      ...                    # previous code
      my $file = new File
      ...                    # initializes values
      display ( $file->bp_value )
      ...                    # previous code
}
...
```

Pseudocode 2: Implementing –bp Value

To implement a solution to the right-click problem, we add an event handler. In doing so, when a user right-clicks on one of the processes along the top of the screen, a dialog box will pop-up and ask the user for a new –bp value. The user can then adjust the –bp flag and have the visualization tool redraw each file accordingly (see Pseudocode 3).

```
...
sub canvasClick {
      ...                    # previous code
      if ( $file_clicked ) {
            $file->bp_value = getBP()
            updateCanvas()
      }
      ...                    # previous code
}
...
sub getBP {
      # make dialog box
      # get user input
      # check and return user input
}
...
```

Pseudocode 3: Implementing Right-Click

Assigning Colors to Processes

Currently, the visualization tool simply initializes each process to a random color (see Pseudocode 4). To some extent, this helps distinguish one process from another. The problem with the current implementation is that each time the visualization tool starts, some processes initialize to the same color or to various shades of the same color (see figures 8-9).

```
...
sub initState {
      ...                    # previous code
      $process->color( randomColor() )
      ...                    # previous code
}
...
sub randomColor {
      # get random color
      # return random color
}
...
sub drawProcess {
      ...                    # previous code
      my $color = $process->color
      ...                    # previous code
}
...
```

Pseudocode 4: Current Implementation of Assigning Colors

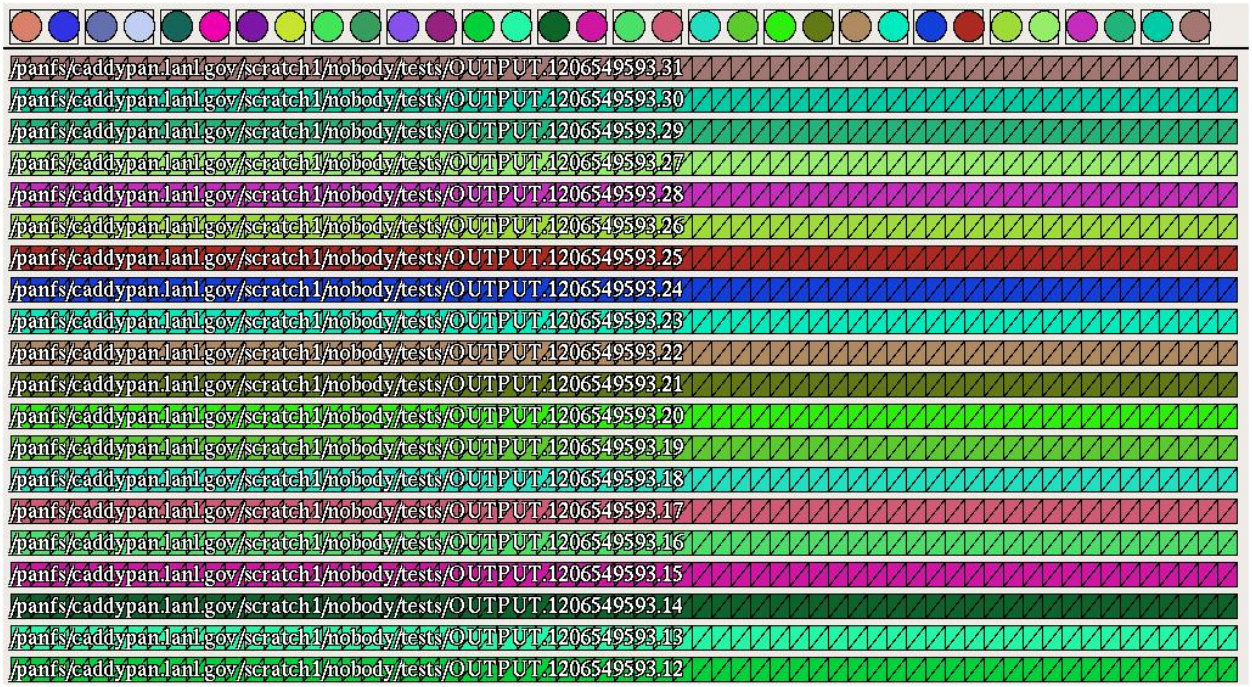

Figure 8: Comparing Colors #1

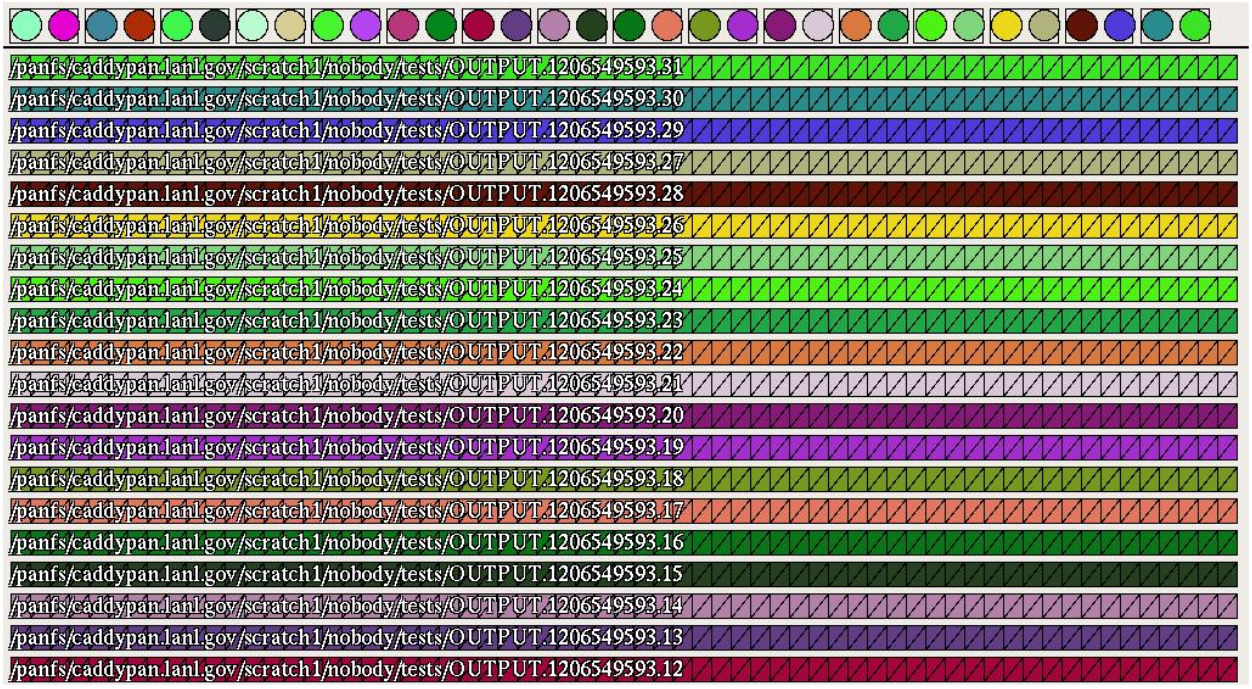

Figure 9: Comparing Colors #2

As a suggestion by our client, it is perferred that the color scheme be uniformly distributed among the number of processes. An example would be, given seven processes, the color scheme can be uniformly distrubuted from the color spectrum (see figure 10). Should there be more then seven processes (which there are), the color scheme automatacally calculates and assigns each processes' color respectively.

Figure 10: Example of Color Scheme

Approach to Assigning Colors to Processes

To implement the color scheme, we need to add another subroutine that determines how many processes are loaded. It will then have to calculate the color values such that the colors are uniformly distributed. Finally, it will have to convert the color values into it respected value so that Perl/Tk can understand (see Pseudocode 5).

```
...
sub initState {
      ...                    # previous code
      $process->color( getColor() )
      ...                    # previous code
}
...
sub getColor {
      # get number of process
      # calculate uniformed colors
      # convert integer values to RGB values
      # return color
}
...
sub drawProcess {
      ...                    # previous code
      my $color = $process->color
      ...                    # previous code

}
...
```

Pseudo Code 5: Implementing Assigning Colors

Project Implementation

The requirements for the project outlined by our client were all independent of each other so we decided to split the requirements and work simultaneously on them. We were able to work well together bouncing ideas off each other when we ran into difficulties because we were not waiting for the other to complete some task.

We had to write one function together before we could get started on the other requirements. Since we were dealing with the output of the software we needed to implement a redraw function that would refresh the screen if a parameter changed. This was a simple implementation; all we had to do was loop through all the defined events and call the drawEvent function that would redraw the events on the screen with the current parameters.

Wrapping Large Files Implementation

The implementation of the file wrapping feature proved very difficult over the course of the project. The final implementation of the feature was different from our original plan. We were not able to get the other files to shift down if a file above it hit the edge of the screen. This would lead to overlapping of the files when they started to wrap. Instead we implemented the wrap feature as an additional flag that could be set so if there was only one output file as in the N-1 case you could select the wrap option. We were able to implement an additional feature to auto-fit the flag size to the width of the screen. This solved the issue when there were many files that needed to be wrapped.

Each time an event occurs that needs to be drawn, a subroutine is called, drawEvent. The implementation of the wrap feature occurs all within this function. The position of the shape drawn is determined by the offset of the event from the trace file. This offset is mapped to an x-position on the screen then drawn. The wrap feature uses a loop to check if the offset is beyond the width of the window. If it is, then it subtracts from the x-position until it is within the window. The number of times the loop runs determines how many lines to shift the event down on the screen, this is done in another for loop (see Pseudocode 6).

```
...
drawEvent{
      ...                    # previous code
      startx = start position
      endx   = end position
      if ( wrapFlag ) {
            while ( startx > width of screen ) {
                  startx -= width of screen
                  endx   -= width of screen
                  count++
            }
            for ( 1 to count ) {
                  yPosition += nextLine
            }
      }
      ...                    # previous code
}
...
```

Pseudocode 6: File Wrap Implementation

The drawn events are based off of the offset from the trace making it easy to translate those positions to new lines in the program. This feature integrates well with the pixel flag since the drawEvent function is called when we redraw the screen the files get wrapped correctly based on the bytes-per-pixel size. The output of the wrap feature can be seen in Figure 11, this is a much cleaner output on the same data set than Figure 2 show before.

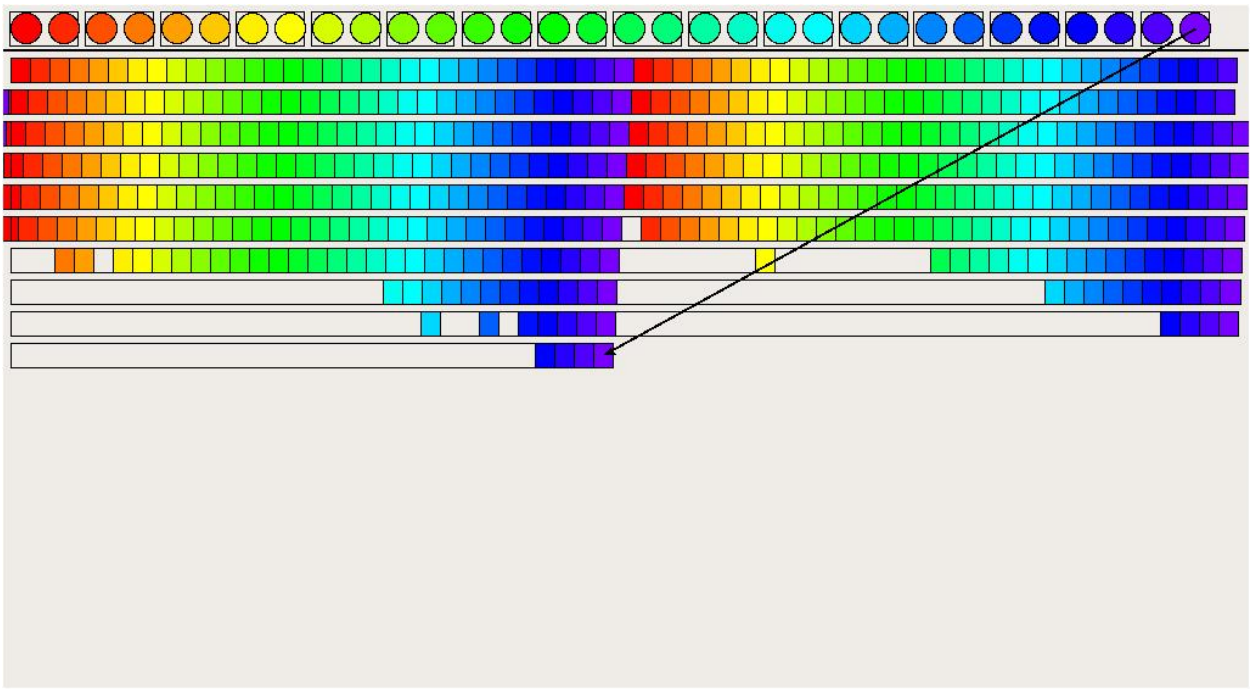

Figure 11: File Wrap Implemented on N-1 Strided Set

Setting the Bytes-per-Pixel Flag Implementation

The implementation of the bytes-per-pixel flag was fairly straightforward, the most difficult part was adding the code to capture the click event and setup the dialog box to change the flag. There was already code implemented to capture a left-click event so we used that to mirror the right-click event. When the user right-clicks in the canvas a dialog-box appears that allows the user to set the bytes-per-pixel flag to a predetermined value or a custom value. The program checks to see if the user has clicked on a particular file, if they have, then the user can set that flag individually otherwise the change is applied to all the files. We added a bytesFlag parameter to the file struct so that they could be set individually. If the user selects apply to all, then the program loops through all the file structs and sets them to the same value. The flag value is hard to determine if you are not familiar with the program so we added an autoSet function that would set the flag value to some predetermined values. We added three functions to the program to implement this feature: rightClick, autoSetFlag, and setFlag (see Pseudocode 7).

```
...
rightClick {
      # create popup menu

      # add options to menu
      Auto, 25%, 50%, 75%, Custom, Apply to All

      # call set function based on option
      setFlag() or autoSetFlag()
}
...
setFlag {
      # check what file was clicked

      # display dialog for parameter
      file->flagValue = parameter

      redrawAll()
}
...
autoSetFlag {
      if ( apply to all ) {
            for ( all files ) {
                  file->flagValue = parameter
            }
      }

      redrawAll()
}
...
```

Pseudocode 7: Bytes-per-Pixel Implementation

Figures 12-13 show the format of the dialog box as well as the result of changing the flag size. There are many other options that appear on the dialog box, these options are additional features that were added to the program and will be discussed later.

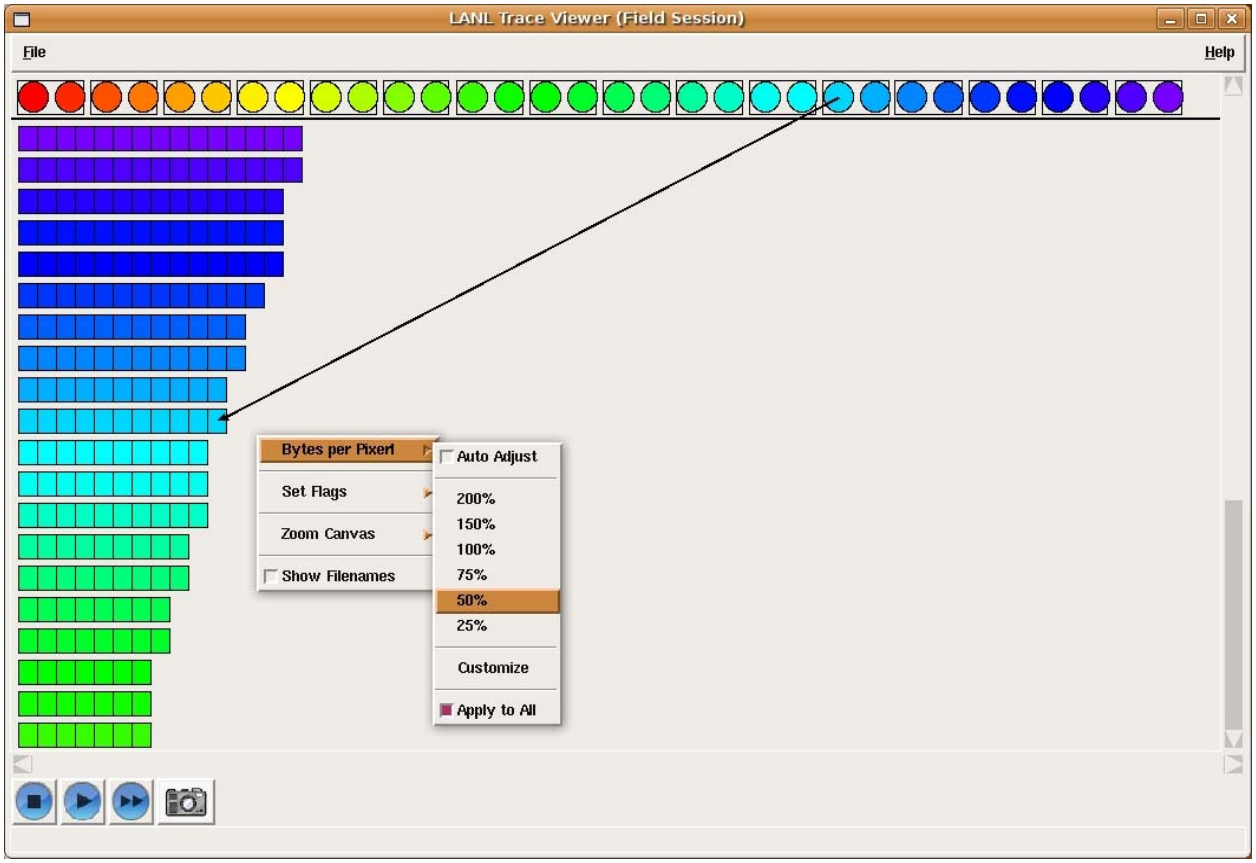

Figure 12: Adjusting Bytes per Pixel Before

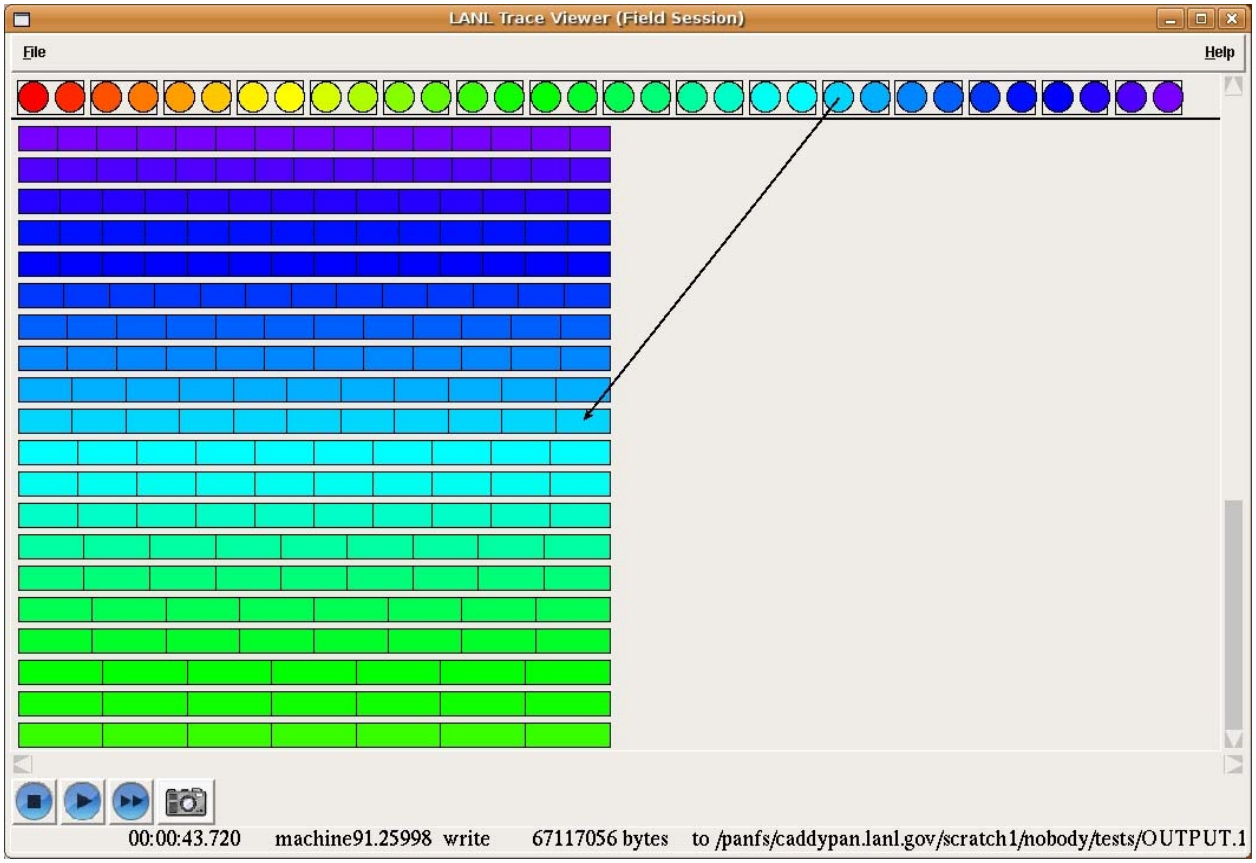

Figure 13: Adjusting Bytes per Pixel After

Assigning Colors Implementation

The implementation of the new coloring sequence required us to come up with a formula for determining a gradient of RGB values based on the number of processes that were in the trace. We wanted the coloring sequence to always cover the spectrum so the values had to depend on the number of processes we had. There is no built in function for assigning gradient colors in Perl/Tk so we developed a nested if/else tree to determine what each color value should be assigned. The setColor function sets up an increment variable to determine how much to add each time, this is based on the number of processes in the trace. We started with red and then move through the spectrum to violet. Each time we set a color, we check what the current color is and based on that we add or subtract the step size from the appropriate color, Red, Blue, or Green. Pseudocode 8 offers a clear view of how the colors are assigned and Figure 14 shows the results of the coloring scheme.

```
...
setColor {
      # calculate step
      step = ( 255 * 5 ) / number of processes
      R=255, G=0, B=0

      # determine color
      if ( R=255 & G<255 & B=0 )
            if ( G + step > 255 ) then G=255
            else G += step
      elseif( R>0 & G=255 & B=0 )
            if ( R - step < 0 ) then R=0
            else R -= step
      elseif ( R=0 & G=255 & B<255 )
            if ( B + step > 255 ) then B=255
            else B += step
      elseif ( R=0 & G>255 & B=255 )
            if (G - step < 0) then G=0
            else G -= step
      elseif ( R<255 & G=255 & B=255 )
            if ( R + step > 255 ) then R=255
            else R += step
      else
            # color does not exist
}
...
```

Pseudocode 8: Gradient Coloring Scheme

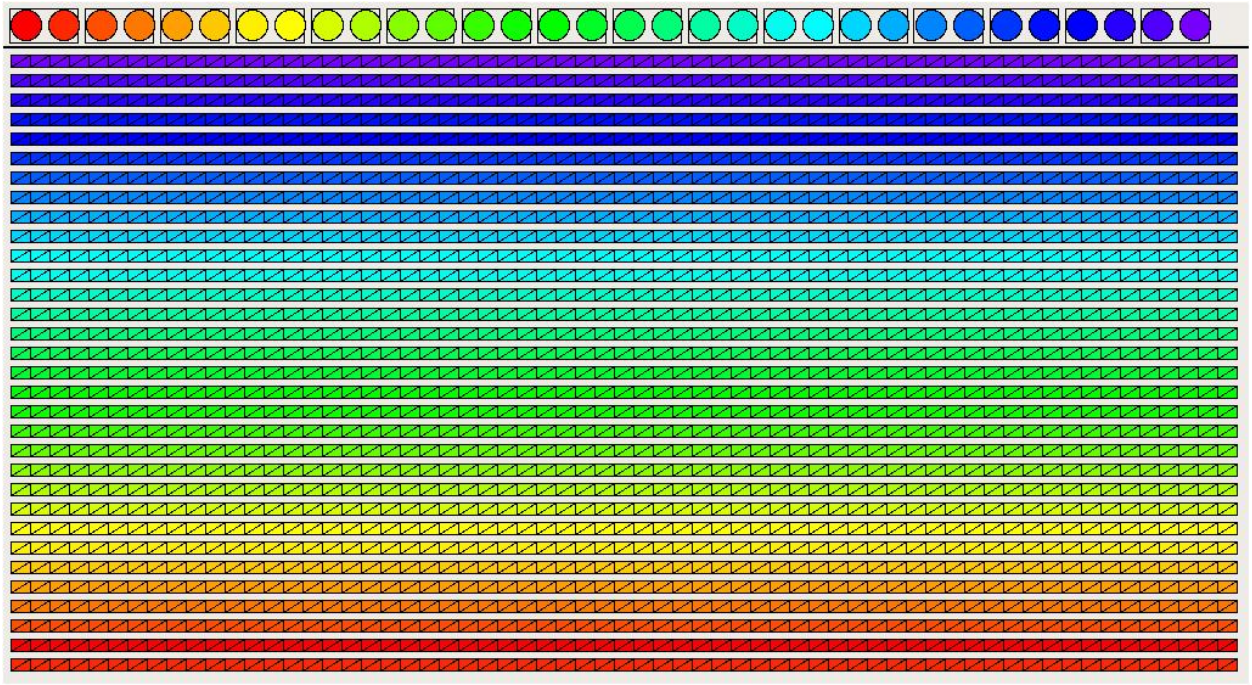

Figure 14: New Coloring Scheme

Additional Implementations

Our requirements allowed us to change broad features of the program which enabled us to add many additional features that added to the functionality and usability of the program along with the features we implemented. Some of the features are not complete and still have a few bugs but we hope that they provide a starting point for future development and ideas for the program.

Since we added the structure to change the bytes-per-pixel flag, we also added the option to change all of the program flags at runtime. Here you can change what files are recognized, the file height, file gap, process diameter, and more. Some of the changes at runtime do not work properly like the file height, but if this feature wants to be added in the future, the code and framework is already there.

In our client meeting he identified that it might be beneficial to add an apply-to-all feature when changing the flags, so you wouldn't have to click on every file to change the size while the program is running. This feature works for all of the flags that you can change at runtime and improves performance by only redrawing the screen once.

In order to get around our trouble with the screen wrap, our client suggested we could add a feature that automatically set the bytes-per-pixel flag so that it fit in one screen perfectly. When setting the flag manually it is hard to tell what the output will be. With the auto fit feature we know it will fit to the screen width perfectly. In the future this feature could be extended to occur automatically when the file reaches the edge of the screen. Currently the user has to tell it to happen.

Our client identified in our meeting that it would be helpful if there was a zoom feature that the user could use to examine certain files closer. We began to implement this feature at the end and it is incomplete. The simulation while in zoom mode does not work properly and you must zoom out and redraw the screen to get the correct display. This would be a good area for future development on the project.

In the original version, the names of the output files were displayed over the visualizations. This made it hard to see what was happening early in the trace and you could not click on the areas covered by the labels. We added a simple feature that can toggle the labels on or off while running the simulation.

An additional glitch with the program occurs during the load sequence. The simulation buttons are enabled but the program does not run correctly if you start it before it is finished loading the trace file. It would be beneficial to disable these buttons until it has loaded correctly. We began to look at this issue and had trouble disabling the functionality of the buttons before the load is complete.

Conclusion

Overall this has been a very satisfying project. We were able to provide our client with quality enhancements to the software that he requested. These changes will be very beneficial to the future development and usability of the program. We were also able to provide additional features and feedback that will lead to additional enhancements and development in the future. Not only was this

project beneficial to the client but it allowed us to expand our skills as computer scientists. We were able to gain good experience using another language, Perl, along with a graphical interface extension, Tk. The learning curve on these tools was fairly quick so it did not hinder us in creating a quality product. We were also able to develop software for an interesting area of computer science. Through the enhancements to the software we made, we have learned how research is conducted on large computing clusters and parallel computing. We hope that this project can continue and provide future students with the opportunity work in an interesting area of computer science and gain new skills.