Ruprecht-Karls Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Bachelor Thesis

Performance Visualization for the PVFS2 Environment

Name:	Withanage Don Samantha Dulip
Supervisor:	Prof. Dr. Thomas Ludwig
Enrollment Number:	2371809
Date of Submission:	November 5, 2005

Ich versichere, dass ich diese Bachelor-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

....

Abgabe Datum: November 5, 2005

Abstract

This document describes how to use the functionalities of SLOG2[1] trace file format in order to facilitate complex visualization concepts in parallel environments. How to analyze trace files generated by parallel I/O activities using MPE [2] for PVFS2 [3] is the basis of this analysis. Implemented set of utilities help improve the user defined performance analysis and visualization of SLOG2 trace files.

Chapter 1 takes an overview of the parallel file system, tracing libraries, logging formats, and visualization programs. Chapter 2 specifies the needs in visualizing the PVFS2 servers and clients together in order to do performance measurements. Chapter 4 describes the used design methods. Chapter 5 describes some of the implementation methods used to achieve the specified goals. An example illustration using the implemented tools is included in chapter 6. Chapter 7 describes the conclusions. Chapter 8 and 9 respectively describe the future works and possible future enhancements.

Key Words : Parallel systems, Interval file format, Trace visualization, Parallel Virtual File System, Scalable Log File Format, Multi-process Environment, Jumpshot, MPI

[1]Scalable log file Format
[2]Multi-process environment
[3]Parallel Virtual File System

Contents

Chapter 1

General goals of thesis

Post mortem analysis of trace files has become a well-accepted technique in performance analysis in current parallel program environments. MPICH2[1] is one of the widely used MPI[2] implementations that use MPI semantics to optimize the communication activities in parallel programs. MPE2[3] which is included in MPICH2 offers a number of performance analysis tools based on post processing approach. PVFS2[4] is a parallel file system which is tailored specially for using in parallel computer environments.

Main goal of the whole project is to visualize the PVFS2 servers using the tools in MPE2. In terms of PVFS2 server behavior analyzation lots of enhancements are needed although the basic functionalities already exist. Most tools have to be modified because of the complexity of activities which occur during PVFS2 I/O. The goals in the scope of this thesis is to determine and implement tools that address the visualization issues on PVFS2 servers.

Most of the modifications that should be done in the MPE2 tools are common needs in the trace analyzation processes. Therefore the tools have to be flexible enough to address general needs in the parallel program trace analyzing community.

[1]A portable implementation of MPI
[2]Message Passing Interface
[3] Multi-Processing Environment
[4]Parallel Virtual File System

Chapter 2

System Overview

2.1 The Parallel Virtual File System PVFS2

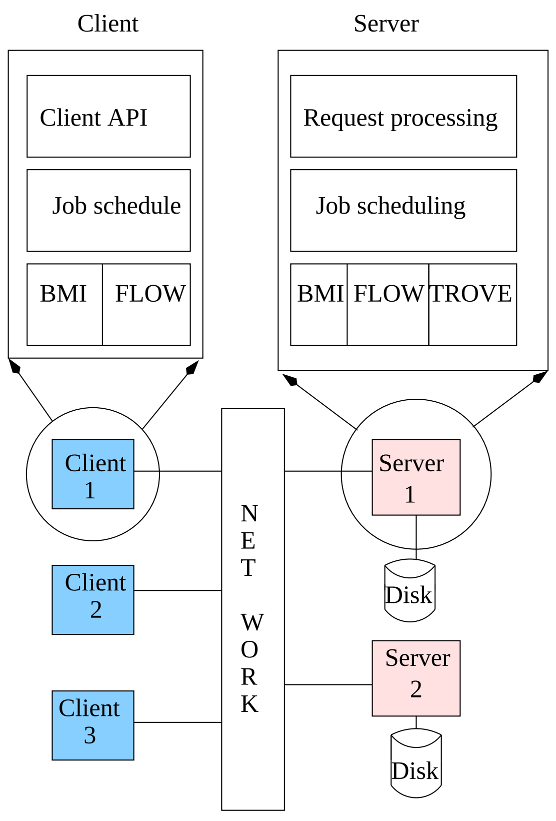


Figure 2.1: PVFS2 software structure

Large parallel computer systems, also called computer clusters are now widely used for scientific computation. In these computer clusters I/O subsystem consists of many disks in number of different nodes. The software that organizes these disks into a coherent file system is called a "parallel file system". Applications can access files that are physically distributed among different nodes in the cluster using the parallel file system software.

Parallel Virtual File System, also called PVFS2 is one of the most popular parallel file systems designed for efficient reading and writing of large amounts of data across the nodes. To achieve this PVFS2 is designed as a client-server architecture as shown in the figure 2.1.

PVFS2 Server runs on nodes that store either file system data or meta-data. According to the type of storage, servers can be categorized into I/O servers and meta-data servers. I/O servers store data in a round robin manner, typically striped over multiple nodes using the UNIX file system. Meta-data servers store all the information about files such as permissions, time stamps, directory hierarchy, and distribution parameters in a Berkeley DB database. A computer node can be configured as either a meta-data server, an I/O server, or both at once.

PVFS2-client Clients run on nodes that read and write the data from the PVFS2 servers. They are implemented as user space daemons. At least one client must run on a node that wishes to access the Parallel Virtual File System.

low level interfaces PVFS2 contains several low level interfaces for performing various types of I/O. Some of the important interfaces are BMI, TROVE, FLOW, and Job scheduling layer.

Buffered Messaging Interface (BMI) provides a nonblocking network interface which is used in file system servers and clients. Currently BMI modules exist for high performance network fabrics such as TCP/IP or Infiniband.

Trove storage object This interface provides access to local files and databases. Trove storage objects, called data spaces consist of byte streams and keyword/value pairs.

Flow interface combines the functionalities of BMI and TROVE in a single transfer and is also responsible for buffering and datatype processing.

Job scheduling layer is a single API that binds all the other I/O interfaces such as BMI, Flow, and Trove. This layer makes scheduling decisions and it is also responsible for thread management.

For more detailed information on PVFS2 refer to [5], [6] and [7].

2.2 The Concept of Tracing

Event tracing is one of the most important ways to analyze program behavior in order to debug or determine the performance bottlenecks of a program. Storing of time-stamped events (e.g. entering a function or sending a message) during the runtime of a program into a log file is generally understood as tracing. Trace files can be fed into graphical programs for visualization and analyzation processes. Some of the well known trace analyzation programs are Jumpshot [3] and Vampir [8].

A simple trace file consists of time-lines of event records . An event record is a timestamp and some data describing some information. A collection of event records that belongs logically together (e.g. events for a process or processor) is identified as a time-line. Trace-data organized in that way can be displayed as GANNT charts. A GANNT chart for a trace

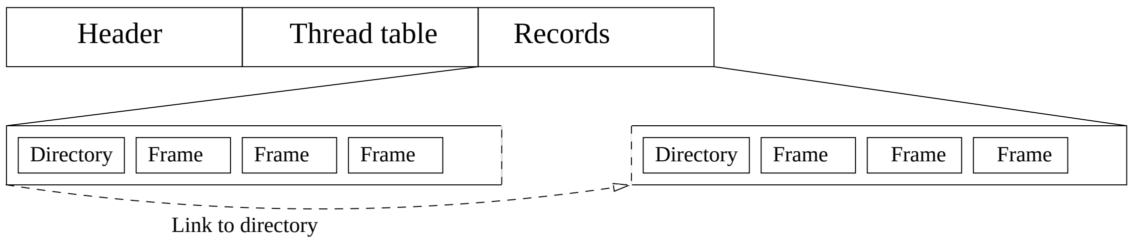


Figure 2.2: Structure of an self-defining interval file

file consists of x-axis as time and y-axis as time-line number. Most of the current available visualization techniques use GANNT chart approach.

Event records are not powerful enough to display performance data.The reason for this is that an event record is only a time-point in the time-line thus making it useless for complex analyzation. Therefore trace libraries and visualization programs use interval records instead of event records. An interval record is an event with an extra field to indicate the duration of the event. In the simple case, an event starts, continues for a period of time and ends. Then an interval record is generated for type complete. This approach is not always applicable. For an example interval record of a thread usually doesn't have a constant time-line for beginning, continuation, and end. When the thread is de-scheduled during its execution, it is only pieces of interval records that represent the thread.

Activities in the nodes are locally stored as interval trace files when an application is running. After the application ends all the trace files from different nodes are merged into a single trace file. Considering the critical resources in a parallel environment such as network bandwidth above approach is one of the best alternatives. This trace file is later fed into the visualization program for analysis.

Trace files can be large, if the capacity of the parallel environments and the number of different activities that are profiled are large. Profiling them and handling them such as merging or manipulating the contents can consume a lot of time and resources.

Therefore special file formats are designed to support trace files with large amounts of data. They can be categorized into pre-defined file formats and self-defining file formats.

In a pre-defined file format, the record format is fixed. That means the length of the data records cannot be changed. In an Example, a record can contain 6 integer numbers and a 12 character string. To add a record type of 7 integers and one 12 character string into this file format is impossible. With pre-defined file format it is easy to parse different files together, but it is not easily possible to add more fields.

Self-defining data format only describe how a valid data record syntax looks like but not the way the data is stored in file. Adding different new record types is easy in this format. A valid file is simply a trace file which follows the data record syntax. As shown in picture 2.2 multiple frames of data can be defined for easy access using the directory structure.

Multiple frames can be accessed only by reading the beginning of the frame with the help of self-defining file format making it unnecessary to read the whole data ahead of them. This approach help to solve the problem of accessing large trace files by defining multiple frame directories for different frames in the file. [1] An abstract tracing approach in interval based self-defined file formats is illustrated in the figure 2.2.

2.3 Tracing in MPI via MPE

MPE is a complete software package, consisting multiple trace analyzing programs. MPE profiling library was developed for MPI [1] in MPICH[2], but can also be used in any other MPI implementation. It provides useful facilities for different levels of trace handling such as creating trace files, visualizing and debugging. A convenient way to do this is to link MPICH with MPE library which will profile MPI activities in the given program.

Users can select automatic profiling or customized profiling for the MPI application tracing. Mored detail on profiling methods see the ”profiling libraries” section in User's guide for MPE [2].

MPE defines and supports several trace-file formats. Two of the most important and widely used trace-file formats are CLOG2 and SLOG2. CLOG2 is an event-based file format (see section 2.2 on event-based file formats) which is designed for profiling in MPI with a low performance overhead. SLOG2 stores its data in a self-defining interval file format. (see section 2.2 on interval file format)

2.4 Jumpshot

2.4.1 Introduction

Jumpshot is the most advanced graphical performance visualization program that is included in MPE. It is written in java swing and understands the trace files from SLOG2 format (refer to section 2.5 on SLOG2). Following is a brief explanation of the main windows of Jumpshot.

When invoked, Jumpshot has a main window which controls the other sub windows. Included sub windows are log conversion window, time-line window, Infobox, histogram, and legend window.

Log conversion window This window allows the transformation of special log-formats into SLOG2 which is understandable by Jumpshot. Currently the conversion of UTE, CLOG, CLOG2 and RLOG is supported. The log file types mentioned above are mostly specific to special needs of profiling (e.g UTE is used in IBM clusters) therefore almost no documentation is available on them. After these formats are converted into SLOG2, they can be viewed by Jumpshot's time-line window.

Time-line window This window is the most important graphical window in Jumpshot where the complete trace file is visualized. All the trace-data is displayed as a GANNT Chart in order to support a good analyzing approach. In this window there are general analyzing

[1] Message passing interface
[2] one of the portable implementations of MPI

facilities such as search, scroll, or zoom. Jumpshot time-line window can be seen in figure 2.3.

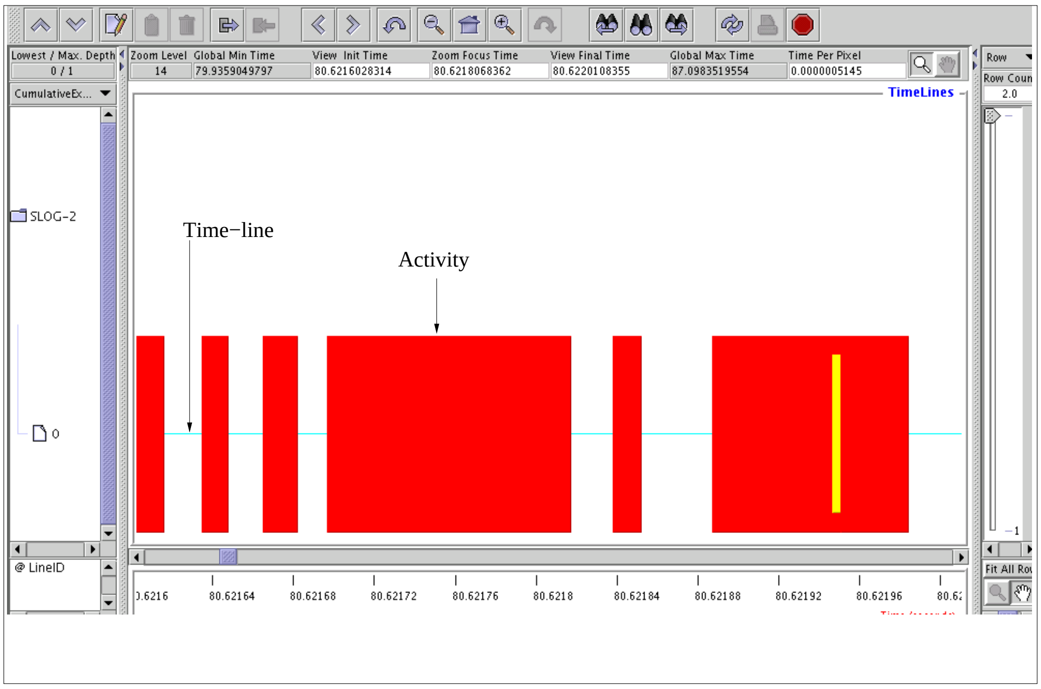

Lowest / Max. Depth: 0 / 1
Zoom Level: 14
Global Min Time: 79.9359049797
View Init Time: 80.6216028314
Zoom Focus Time: 80.6218068362
View Final Time: 80.6220108355
Global Max Time: 87.0983519554
Time Per Pixel: 0.0000005145
Row Coun: 2.0

Figure 2.3: Time-line window

The legend window shows the description of the different activities that are used in time-line window. In legend window the user can change different characteristics of the different categories [3] such as search-ability and visibility. Different colors are defined in the legend window to identify different activities in the time-line window. As an example, see the job activity defined in legend window in figure 2.4. In the time-line window in figure 2.3 we can identify the job activity in the same color. User can also change the colors in the legend window that will also change in the time-line window accordingly.

Histogram window A histogram window is also supplied to get more specific information on a user-selected time interval of the time-line. This window helps to analyze specific statistics in the area that was selected from time-line window.

Infobox Each event in the time-line can be further analyzed with the Infobox window that takes user specified information as a pop-up window. This window is helpful to visualize extra data which was generated from the tracing API. Helpful information such as duration of an operation or server id can be traced in order to be used by Infobox.

2.4.2 Why Jumpshot?

Jumpshot visualizes SLOG2 file format which is specially defined for trace files in MPE environment. One of the most influential aspects of Jumpshot it that its ability to handle

[3]the different activities identified by SLOG2. e.g. MPI_SEND and MPI_SENDRECV

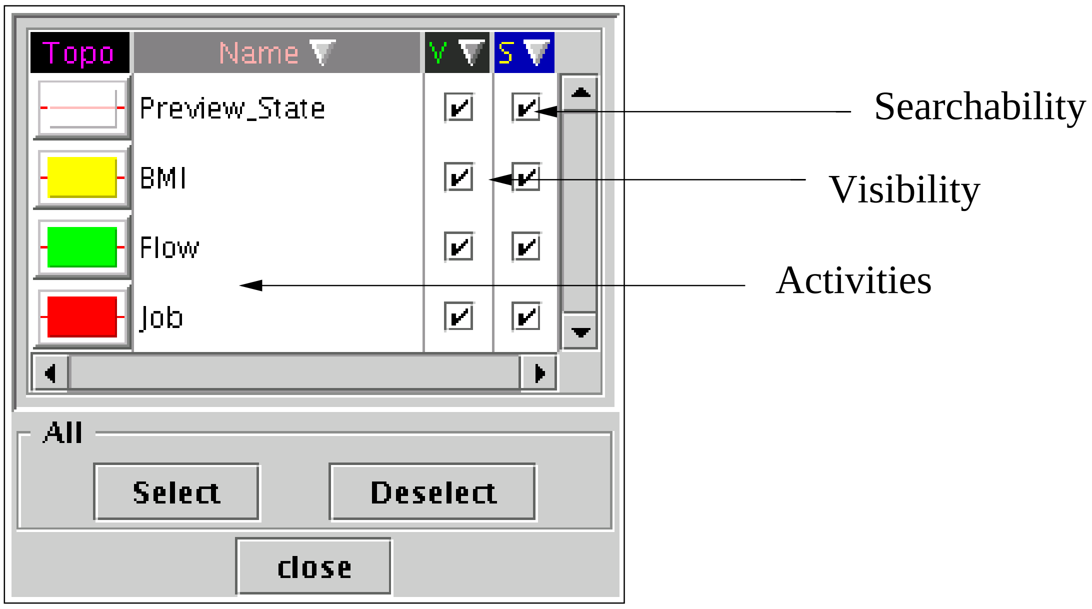


Figure 2.4: Legend window

trace-files which can be extremely large[4]. Handling large trace files is necessary, regarding the complexity in visualizing PVFS2 environments. As an example the number of nodes in PVFS2-installed large clusters can vary from several hundreds to numbers of several thousands. [3]

2.4.3 Limitations

Take a parallel environment where MPICH, MPE, and PVFS2 is installed. We can get the trace files of the MPI communication from the PVFS2 clients in that environment. These trace files can be directly viewed in Jumphot. It is also necessary to see the activities which occur in the PVFS2 servers for performance tuning purposes.

2.5 SLOG2

SLOG2 is a self-defining interval trace file format used in MPE.

In the figure 2.5 we see a possible ordering of the SLOG2 format. Note that the SLOG2 API describes only a possible representation of objects in SLOG2 file. This means that any file which is organized according to this order can be recognized by the SLOG2 API as a valid file.

Different parts of a SLOG2 file is briefly described here that are discussed in more detail in the SLOG2-Draft [4].

[4]according to authors, Jumphot-4 can handle extremely large files reaching gigabyte limit. [3]

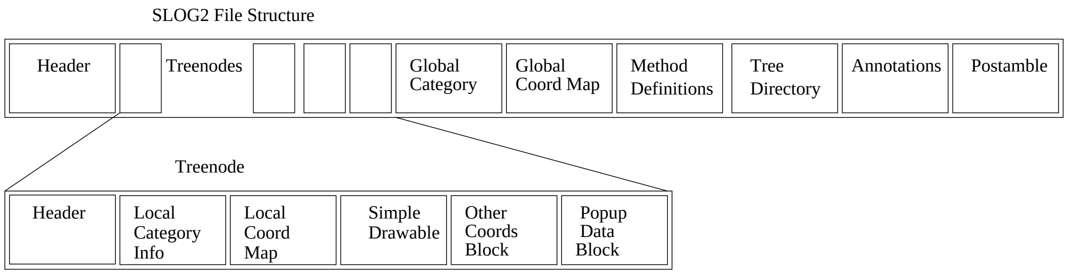


Figure 2.5: SLOG2 file structure

Header Information about the file is stored in header. Possible information are SLOG2 version, and other meta-data information.

Global category info Different categories used in the SLOG2 file are described in Global category info. As an Example we can identify the different MPI calls which were profiled as categories. A category contains information about the activities such as name, color, and meta information fields. Category information is directly read by the legend window, which was described in section 2.4 on Jumpshot.

Global coord map A table in which y axis is mapped to different time-lines is called the global coord map. It contains the title of the map, column id and label id. A sample global coord map is shown below.

colum	lineID	IineID
1	0 :	0
2	1 :	2
3	2 :	1

Table 2.1: Global Coordination map

In the above map, all the drawables which have the line id 0 will be mapped to first time-line. Drawables with time-line id 1 will be drawn in 3rd time-line and etc.

Method definition Information Different methods which can be defined in order to specify pop-up data in the Infobox. Visualizing methods is in the primary states of development in Jumpshot.

Tree directory The drawable objects in SLOG2 are ordered as tree data structures. Tree directory defines offset for each treenode from the beginning of the SLOG2 file and the start and end time.

Annotations User defined information and viewing history is stored in annotation part.

Postamble The locations of the global category info, global-coord-map, method-definition, tree-directory, and the annotation blocks are stored here in 4 byte-intger values.

Treenode Treenode is divided into different blocks for better identification. The smallest treenode, called a simple drawable will be discussed here.

Header Meta-data information about the other parts of the treenode.

Drawable A drawable has a fixed length structure. It consists of the start and end times, category id, y coordinate, and offsets into the next two blocks.

Other coords block This is an extra block to describe the other coordinates for complex descriptions about the drawable.

Pop-up data block Data for the pop-up methods, used by Infobox in Jumpshot which was discussed in section 2.4.

Drawables can be further categorized into two groups called primitive and composite.

2.5.1 Primitive drawable

Primitive drawable is the smallest element visualized by Jumpshot. This is an instance of the drawable. Currently three different types of primitive drawables are supported. Event, State and Arrow are the three of them and they are the basic elements in the SLOG2 file.

2.5.2 Primitive drawable types

Event is a one time-stamped point in the time-line, which consists of a time-line id and a time-stamp. Events are drawn in time-line canvas in Jumpshot as a vertical line.

State is defined by two time-stamps that are inside one time-line. States are drawn in Jumpshot as a rectangle.

Arrow is also defined as a state with two time-stamps, but not necessarily in same time-line. Arrows are often drawn between states and are mostly used to show the relationships between two different states.

2.5.3 Composite drawable

Composite drawable is a collection of primitive drawables with an optional category and optional text string. It is easy to combine lots of primitive drwawables which can be grouped into one logical unit with the concept of a composite drawable.

As an example, take all the BMI activities that occur during the application run-time. We can profile the name of the server as extra information during for BMI activity. Later the BMI activities from one server can be grouped into one composite drawable. Then we need only to refer to the composite drawable other than referencing to all the primitive drawables to address all the BMI activities of a specific server.

Chapter 3

Specification of the Goals

3.1 Requirements

In order to analyze the PVFS2 file system behavior, it is necessary to profile specified activities in the servers and clients for performance measurements.There are lots of communication activities in the high level and low level interfaces when the servers are running. Not only the number of types of different communication activities are large but also the number of their occurrences. Therefore they generate a lot of overhead when profiling all of them. After the trace-data from servers and clients are included in one trace file, there are lots of activities that are mostly overlapped. Large number of activities make the visualization useless because they fill up the display of the graphical program and make the debugging and analyzation nearly impossible. Therefore a thorough study is needed in order to specify which activities should be profiled into the traces according to specific needs.

When an application ends there are two trace-files. One for the PVFS2 client-server activities and one for the MPI communication. These two trace files have to be merged in order to analyze the whole system behavior for the application. This should be done with the help of a merging tool.

Because of the tight association of the servers and clients and the names of the interfaces are mostly the same, it is extremely difficult to differentiate servers and clients. Therefore after merging there should be methods to identify the server and client activities.

Because of the overlappings much of the communication is not easily visible in the graphical program. The problem of overlapping have to be solved in the mean time addressing the problem of server-client identification. As client-server activities and MPI activities are related to each other, a general concept of identification must be implemented to address the overlapping problem. How the overlapping should be dissolved is illustrated in the figure 3.1.

In the figure 3.1 there are two large overlapping activities. They have a number of small activities inside them. As they are in one time-line we cannot see some of the hidden small activities. Small activities become visible after the large activities are ordered into two time-lines without the overlapping.

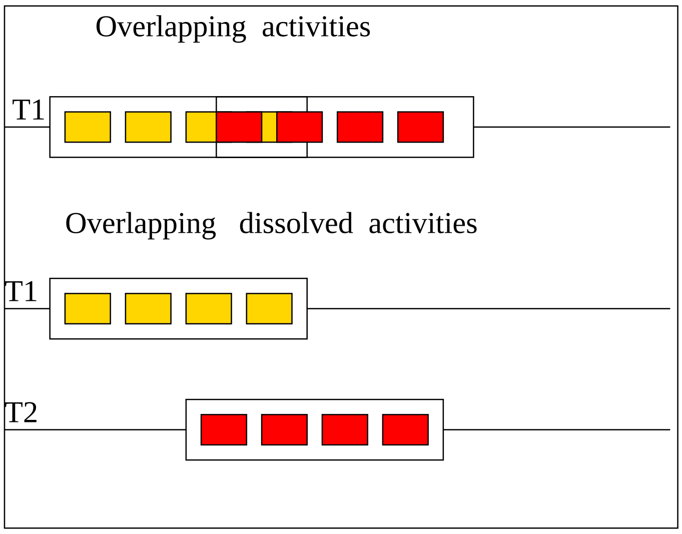


Figure 3.1: Approach on overlapping dissolving

3.2 Representation

GANNT charts are one of the famous and effective ways to visualize the parallel program activities as discussed in Chapter 2.2. MPE uses events, states, and arrows to represent the MPI communication and MPI I/O activities. Up to some level this approach is adequate to represent the PVFS2 server activities. Most of the server activities can be presented with events and states. Therefore the generated trace files which are in CLOG2 format (an event-based file format in MPE) can be used as the preliminary trace log file format. CLOG2 trace-files should be converted into SLOG2 for better identification and analyzation. The reason for this is that SLOG2 has a well-defined API. [4]

Chapter 4

Design methods

The main designing process can be categorized into two, namely trace management and graphical management.

4.1 Design of trace management

There are two CLOG2 files after a program has ended. The first one is for the PVFS2 client-server activities and the second one is for the MPI activities. Both CLOG2 files are converted separately into SLOG2 format using the converting tool *clog2TOslog2* which is a java-based program that *Clog2TOslog2* is already included in MPE.

Thereafter the two SLOG2 files have to be merged together. This is done by *MergeSlog2* [1]. After the two SLOG2 files are merged, activities such as Job (see section 2.1) can be analyzed to understand the system behavior in detail. It was decided to use the maximum capacities of the SLOG2 to reach this goal because of the flexibility in handling SLOG2 files.

4.2 Design of graphical management

Not moving far away from the goal to implement tools that will address general issues, composite drawable approach to identify activities that belong logically together was taken. After the SLOG2 files are converted into SLOG2 files with composite drawables, it has to be reorganized to make the hidden activities visible. A program called *Slog2ToCompositeSlog2* will be written in order to make composite drawables. Another program called **CompositeSlog2TOLineIDMap** will be implemented in order to resolve the ovelappings in the composite drawables. Time-lines of the drawables can be changed after composite drawable overlappings are dissolved. Therefore it is necessary to find a solution to store the original time-lines and show them accordingly in the new SLOG2 file. Since these two approaches are lying technically close to each other, it was decided to integrate this also to the **CompositeSlog2TOLineIDMap**.

[1]**MergeSlog2** is a java-based log converting tool, which was implemented by Julian Kunkel

4.2.1 Composite generation

It is necessary to identify under which pre-conditions composite drawables have to be generated. All the drawables that fulfill a given pre-condition are put together into one composite drawable.

4.2.2 Rearrange the composites

Overlapping dissolving is done after the composite drawables are generated. Overlapped composite drawables are relocated to another time-lines. All the original time-lines from the SLOG2 file are stored and they will be used to rearrange the drawables to their respective time-lines.

4.2.3 Line Id map generation

In SLOG2 file format, there is a concept called line id mapping. This is for Jumpshot to understand how many time-lines exist in a given SLOG2 file. Therefore every SLOG2 file has at least one default line id map. This is realized as a Hashtable in java. Because of Jumpshot's ability to support multiple time-line id maps, *CompositeSlog2TOLineIDMap* will define more time-line id maps to show the relationships.

One of the main advantages of adding more time-line id maps is that it effectively saves the valuable working-memory space which is needed for the SLOG2 files[2]. Another important aspect is that Jumpshot already supports multiple line id maps making it unnecessary at this level to make changes in Jumpshot. This makes SLOG2 files generated with the tools described here also usable in MPE environments other than that in the development versions in Universität Heidelberg.

4.2.4 Arrow generation

Arrows are mostly used in performance visualization programs to show relationships. They can also be used to emphasize points in the graphical canvas. Using arrows makes the analyzing and manipulation easy in particular cases. Hence arrows are also states, they can also be included in composite drawables providing more flexibility in graphical analyzation process. Therefore a program which can be configured to add the arrows to a given SLOG2 file has to be written for possible future enhancements. A program called *SLOG2TOArrowSlog2* is implemented to illustrate how the arrows work, as a reference for the future implementers.

4.3 Approach

Following picture describes the tracing approach taken to visualize the PVFS2 client-server activities in the MPE environment.

[2]this is important because a different approach like adding old time-line information on every drawable should always be multiplied by the number of drawables which can be really large.

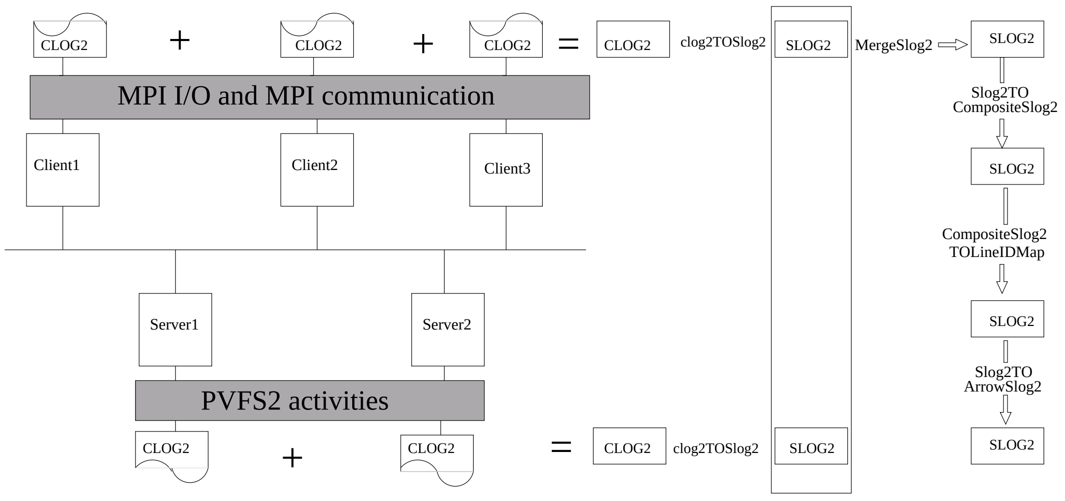


Figure 4.1: PVFS2 tracing approach

4.4 Designing of *Slog2ToCompositeSlog2*, *CompositeSlog2TOLineIDMap*, and *Slog2ToArrowSlog2*

Programming methods of the three programs will be discussed below in detail. Explanations are kept close to the program code in order to help the future code implementers to understand the code easily. Abstract file conversion process can be figured as below as all the implemented programs convert one SLOG2 file to another SLOG2.

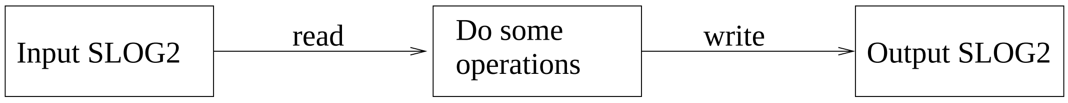


Figure 4.2: Abstract tool structure

4.4.1 General functions

Two functions, which are used in all the programs mentioned above are *readFile* function and the function to write a single drawable into output SLOG2 file called *addDrawable*.

Read SLOG2 file (*readFile*)

After checking for possible exceptions, number of children per node and the leaf size of the original SLOG2 file is read into the memory(Note that the SLOG2 file data is stored as tree-structures for fast reading in Jumpshot). Later these values are initialized for the new output SLOG2 file. Category map, shadow category map[3], time-line id map, and drawables are read

[3]shadow categories are designed for visual enhancements for the smallest events in Jumpshot. e.g. with shadow category we can show a number of tiny drawables from time x to time y with a one large drawable that expands from x to y

from the input SLOG2 file. Note that the drawables are stored in a re-sizable array list[4] in order to simplify the manipulation.

Add drawable (*addDrawable*)

Drawables have to be added to the output SLOG2 file after the the necessary modifications for the drawables in the SLOG2 file are done. This function supports adding a drawable into the output SLOG2 file after checking the time order[5].

4.5 Slog2ToCompositeSlog2

Current *Slog2ToCompositeSlog2* can make composite drawables for a given category (e.g. user can choose which category should be analyzed as the main category). The second option is to build composite drawables according to extra information, which is defined by the profiling process (e.g different identification numbers are given to all the BMI activities that occur during the program runtime, thereby making composite drawables for all the BMI activities that have the same identification number).

Make the composite drawables according to categories (*makeComposite-ForSLOG2*)

makeCompositeForSLOG2 is the main method that generates the composite drawables according to categories defined by command line. The procedure can be divided into further steps as explained below. First the the whole SLOG2 file will be searched for the drawables in the category that will be used as the composite drawable category (from here on called as `major-category`). After finding the drawables for major category, it will be checked whether they have drawables within their boundary. Hereby all the drawables are searched for the category that the user defined for inclusion (from here on `minor-category`).

Find drawables in a drawable (*findDrawablesInADrawable*)

This method takes one primitive drawable from the major-category as argument and search the whole time-line for drawables from minor-categories that are within it's boundaries. Note that if a primitive drawable from minor-category is inside two or more drawables from the major-category, it will be included in the major-category drawable which has the earliest beginning time.

Make composite drawable (makeCompositeDrawable)

After all the minor-category drawables are found for one major-category drawable, the desired composite drawable is built. This is done by *makeCompositeObject()*. Array of minor-category

[4]See java.util.ArrayList in java documentation
[5]check weather begin time is smaller than end time, otherwise gives an error.

drawables are concatenated with the major-category drawable and the composite drawable is generated. Note that building composite drawables has to be done with extreme care due to the possible information loss. An example of generating a composite drawable is illustrated in the implementation section 5.

Add the rest primitive drawables

Rest drawables are the drawables that don't belong neither to major-category nor to minor-category. These rest drawables are also added to their respective time-lines without change during the composite drawable making process.

Add rest drawables from minor category (*makeRestDrawabes*)

This function is defined to add the rest drawables from the minor-category. Remember that the above function also added the drawables which didn't belong to both major and minor-categories. Thus the rest drawables are from the minor-categories and they don't fit in the boundaries of any major-category drawable.

Add all drawables into new SLOG2 (*addAllDrawables*)

All the manipulated drawables are stored in a temporary array list after all the above steps are finished. Note that the temporary array list now contains all the drawables but they are not necessarily in the correct time order due to the insertion of composite drawables and removing of primitive drawables. Because the drawables should be written to the SLOG2 file in the increasing end time order they are ordered according to their end time and written to the output SLOG2 file.

4.5.1 Make composite drawables according to the identification number

Second variation of *Slog2ToCompositeSlog2* is to use the extra information (such as line id number) as the major condition for composite drawable building. All the drawables in a time-line that have the same information will be grouped into one composite drawable. This function is implemented for id number identification in PVFS2 servers and clients, but can also be used for any other relevant information identifying processes.

Find drawables with the same id (*findDrawablesWithSameID*)

Different information values are fed into the Infobox in SLOG2. This information is fetched back to make the composite drawable objects. As the info values are not known early, first a list of all the different info values are generated after searching the whole SLOG2 file. Hereafter all the primitive drawables for each single time-line is compared with these values and the drawables that have the same identification number are grouped into a composite drawable using *makeIDComposite*.

Make a composite drawable (*makeIDComposite*)

After a group of primitive drawables are given, one composite drawable will be generated with a new category. During this process only drawables from the type "composite" are generated unlike the category selecting method. After the composite drawables are generated, they will be written into the output SLOG2 file with the help of the *addAllDrawables* method defined in sub section 4.5.

4.6 CompositeSlog2TOLineIDMap

This program is implemented to dissolve the overlappings in composite drawables and make the necessary adjustments to the line id map. Note that if this program runs on an input SLOG2 file with primitive and composite drawables or only composite drawables, all the overlappings of the composite drawables are resolved. When there is only primitive drawables in the SLOG2 file no change will be done.

Read and insert drawables

For reading drawables from a given SLOG2 file and inserting them into the output SLOG2 file refer to section 4.5 of the *Slog2ToCompositeSlog2*.

Overlapping dissolving

Before dissolving overlappings old time-line id is stored in order to refer to where the drawable belonged. This is because of the possibility of composite drawables moving into another time-lines. Different methods can be used to dissolve this problem. After considering methods which could deal better with SLOG2 files, the *method of a moving point*[6] was selected.

In the moving point method, the end-time of the first composite drawable is selected as the moving point because surely the first composite drawable doesn't collide with other composite drawables. Then it is checked whether the next composite drawable's begin-time collides with the moving point. If the composite drawable doesn't collide with the moving point it will be added to the current time-line and the moving point will get the end-time of the added composite-drawable. With this method all the composite drawables that don't collide with the moving point will be added to the time-line. Then a new time-line will be initialized as a sub line. After defining the moving point for the new time-line the above procedure is run again. This is repeated until all the composite drawables in the time-line are finished. Then this will be repeated for all the time-lines. Primitive drawables are added to their respective time-lines without checking for overlapping.

[6]The author refers this method as moving point method, because a time point which moves from the beginning to the end of the time-line will help dissolve overlappings.

4.6.1 Generation of time-line id maps

A time-line id map is a Hashtable[7] where the mapping of the SLOG2 file is stored. After the necessary information was collected during the overlapping dissolving, a new time-line id map has to be generated for the newly defined SLOG2 file. Remember the input SLOG2 file usually contains only a simple time-line id map describing the number of time lines and the way they are ordered.

The new time-line id map will be designed as a two level time-line id map where we see the original time-line id number and the number of sub lines which were added during the overlapping dissolving.

Designing the new time-line id map is discussed below. For the interested implementers an example on how to generate a time-line id map is included in the chapter 6.

The line-id map consists of rows that represent the number of levels and the column numbers that represent the number of lines in a level. These values are stored in a one-dimensional array simply calculated from the number of raws and number of columns.

Labels	LineID 1	LineID 2
0:	0	0
1:	0	1
2:	0	2
3:	0	3
4:	0	4
5:	1	0
6:	1	1
7:	1	2
8:	1	3

Table 4.1: A two level Time-line ID map

In the figure 4.1 we see an example of a two level time-line id map. In the "lineID 1" column we see the original time-lines of the SLOG2 file. In "lineID 2" we see to how many sub-lines the original time line is extended. As an example the time-line number "0" is splitted into 5 sub-lines.

As we see in the system overview Jumpshot is already equipped with facilities to visualize different time-line id maps. Changing the time-line id maps as discussed above and in specification on chapter 3 is recommended, because adding user defined time-line id maps will help to improve the analyzation possibilities in the same time saving memory.

4.7 Slog2TOArrowSlog2

Currently Slog2TOArrowSlog2 facilitates adding arrows to composite drawables which are lying after each other.

[7] see the JDK documentation on java.util.Hashtable

An arrow is defined as a state drawable that consists of two events. Therefore arrows can be generated from any point of a time-line to another point of a time-line in the graphical canvas.

Before declaring a new arrow drawable in SLOG2, we must specify its characteristics. This information is given by defining the category for the arrow. Therefore before generating the arrows a category is defined with arrow characteristics. How to define a category is illustrated in section 5.

The category is identified with its index. Great care should be taken to avoid possible overlappings of the new index with consisting indexes that were already used. After the category for the arrow is defined, it should be added to the global category map otherwise the drawables will not be ordered according to the category.

Generating arrow drawables according to given speciications will be done after category definition. All the categories are written into the array list where the other drawables from SLOG2 also exist. Then all the drawables are ordered according to their end-time. Then they are written into the output SLOG2 file with the help of *addAllDrawables* method discussed on sub section 4.5.

Chapter 5

Implementation

In this chapter, general issues considering the implementation process are explained. All the programs are implemented in java. Source code for the slog2 software development kit in MPE2 is located under `<PATH_TO_MPE2>/src/<PATH_TO_SLOG2SDK>/src/`

5.1 Get the number of time-lines in the input SLOG2 file

Following function can be use to get the number of time-lines after the time-line id map is read into memory.

```
Iterator itr = lineIDmap.iterator()           //  Initialize a new iterator
while (itr.hasNext())
{
  LineIDMap IDMap = (LineIDMap) itr.next()    //  Read the line id map
}
YCoordMap ymap = IDMap.toYCoordMap()          //  Convert it into ycoormap
num_rows=ymap.getNumOfRows()                  //  Get the number of rows
```

5.2 Category generation

Defining a new category is always needed when generating both primitive and composite drawables. Class Category.java defines the characteristics of categories and is located under slog2sdk/src/base/drawable/Category.java. Following constructor is one of the most convenient from all the available constructors for generating a category.

```
public Category( int idx, String name, Topology in_topo,
ColorAlpha in_color, int in_width )
```

```
index = in_idx;                     // id number for the category
name = in_name;                     // name of the category (e.g. BMI)
topo = in_topo;                     // Topology o=event, 1=state and 2=arrow
color = in_color;                   // ColorAlpha is a sub class of
                                       java.awt.Color with transperency
width = in_width;                   // width of the line
infokeys = null;                    // This is for extra information. This
                                       can be used to set InfoKeys
infotypes = null;                   // infotypes are defined in
                                       base.drawable.InfoType.java
methods = null;                     // see base.drawable.Method.java
summary = new CategorySummary();    // base.drawable.CategorySummary.java
hasBeenUsed = false;                // verify weather the category is used.
isVisible = true;                   // make the drawables of this
                                       category visible in jumpshot time-line
isSearchable = true;                // make the drawables of this
                                       category searchable in jumpshot time-line
```

5.3 Generate a composite drawable

There are different ways to generate a composite drawable as defined by constructors in the bas.drawable.Composite.java. But the following method is encouraged because it lets the info values of the primitive drawables to be included in the composite drawable.

1.Generate an empty composite drawable defining how many primitive drawables will be inserted inside.

```
Composite newComposite= new Composite( int Nprimes );
```

2.Add the primitive drawables into the composite drawable

```
newComposite.setPrimitives( final Primitive[] in_primes );
```

3.Integragte a new category into the composite drawable.

```
newComposite.setCategory( final Category in_type );
```

Attributes of the composite drawable can be changed using the methods defined in basa.drawable.Composite.java

5.4 Time-line id map generation

Line id maps can be generated for simple time line identification or for complex relationships illustration. An example is shown how to generate a time-line id map.

1.Generate a YcoordMap using above values

```
YCoordMap ycoordmap =new YCoordMap(numberofraws,numberofcolumns
,title,columnnames,emap_elems,map_methodIDs);
```

```
int numberofraws;           //  number of raws
int numberofcolumns;        //  number of columns
String title;               //  user defined title for the map(visible in jumpshot)
String []columnnames;       //  user defined names for the columns(visible in jumpshot)
int raw_column;             //  numberofraws*numberofcolumns, values are saved
                            //  in a one_dimesional array
int emap_elems[];           //  values for the table
int number_of_method_ids;   //  number of methods
int []map_methodIDs;        //  method values
```

2.Generate the time-line id map using the above ycoord map.

```
LineIDMap newlineIDmap =new LineIDMap(ycoordmap);
```

3.Add the newly generated line id map to the LineIDMapList

```
LineIDMapList.add( newlineIDmap );
```

5.Write the lineIDMapList to the output slog2 file.

```
OutputLog.writeLineIDMapList(lineIDmaps);
```

5.5 Read the info value

SLOG2 info value can be used for user defined analyzation. Following method can be used to read the info value of a specific drawable.

1.Get one of the drawables from the drawable array list.

```
Drawable newDrawable=(Drawable) drawableArrayList.get(i);
```

2.Get the info value of the position n

```
draw.getInfoValue(n);
```

5.6 General issues

Tools that are discussed here serve mainly in order to manipulate SLOG2 files. All the tools read a SLOG2-file as input and write another SLOG2 file as output. During this process drawables are modified under given specifications. During the rewriting process all the drawables should be inserted according to increasing drawable end-time order. This phenomenon is not visible during the drawable writing process because SLOG2 generation process runs smoothly without generating warnings. It is later visible in Jumpshot or slog2print. (Only a section of the drawables are inserted in the SLOG2)

Chapter 6

Example

6.1 Execute the test MPI program

Before running a program with tracing support, PVFS2 environment should be configured correctly with MPE tracing library support. Because this installation process has to be done with maximum care and some of the software versions are only available in development versions, there is a special installation script attached in the document describing how to install PVFS2 with MPE support.

Test programs can be compiled and run as illustrated in section 6.2 after the installation. The example test program illustrates writing data into a file by many MPI processes.

6.2 Generate the trace files

Compiling the program and generating the trace files are done as in the following example. Assume that the test file ”mpi-write-test.c” is located in `<PATH_TO_PVFS2_SRC>/test/client/mpi-io/` where other numerous test programs from the developers are also located.

```
cd <PATH_TO_PVFS2_SRC>/test/client/mpi-io/
mpicc mpi-write-test.c -o mpi-write-test -llmpe -lmpe
pvfs2-set-eventmask -m /pvfs2 -a 0xFFFF -o 0xFFFF;
mpiexec -np n ./mpi-write-test -f pvfs2://pvfs2/test; // n = number of nodes
pvfs2-set-eventmask -m /pvfs2 -a 0 -o 0;
```

PVFS2 client-server trace file (pvfs2-server.clog2) is generated in `/tmp` directory after the application is finished. MPI activity trace file is generated in `<PATH_TO_PVFS2_SRC>/test/client/mpi-io` as mpi-write-test.clog2. Both clog2 files are converted using clog2TOslog2 conversion tool.

6.3 Convert the trace files into SLOG2

1. `clog2TOslog2 pvfs2-server.clog2`

Figure 6.1 shows the pvfs2-server.slog2 in Jumpshot.

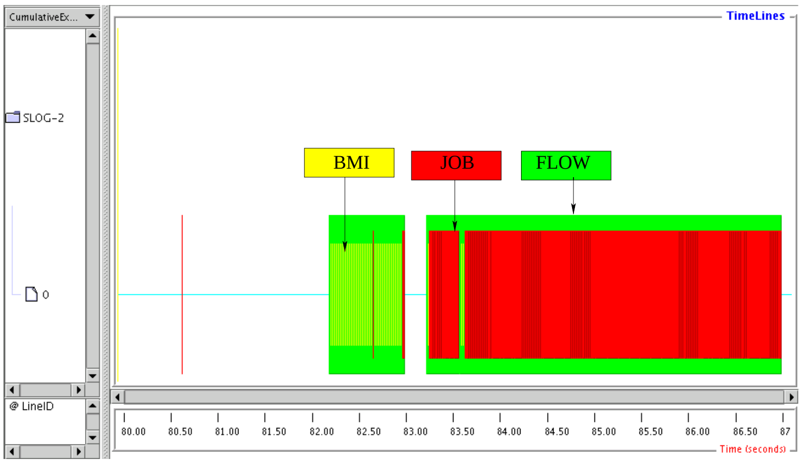


Figure 6.1: PVFS2 client-server SLOG2 time-line window

2. `clog2TOslog2 mpi-write-test.clog2`

Figure 6.2 shows the mpi-write-test.slog2 in Jumpshot.

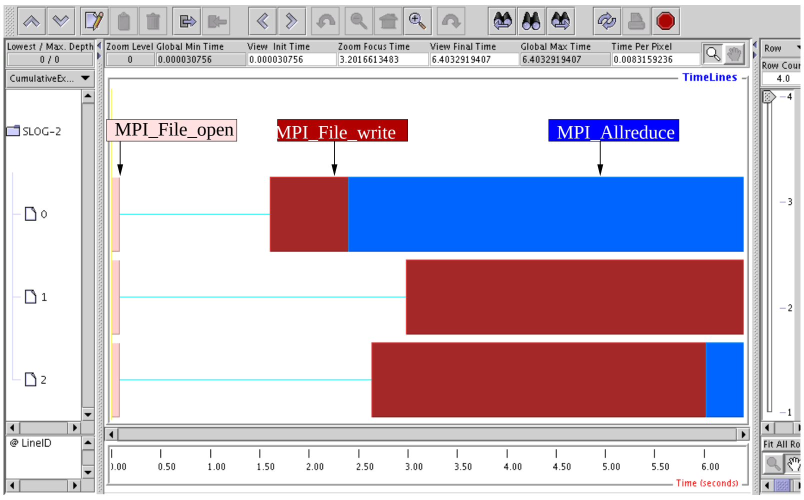

Lowest / Max. Depth: 0 / 0
Zoom Level: 0
Global Min Time: 0.000030756
View Init Time: 0.000030756
Zoom Focus Time: 3.2016613483
View Final Time: 6.4032919407
Global Max Time: 6.4032919407
Time Per Pixel: 0.0083159236

Figure 6.2: MPI activity SLOG2 time-line window

6.4 Compile and run the trace management programs

All the SLOG2 converting programs are copied to <PATH_TO_SLOG2SDK>/src/logformat/slog2/pipe/

6.4.1 Compile and run MergeSlog2

Compile with java

```
javac <PATH_TO_SLOG2SDK>/src/logformat/slog2/pipe/MergeSlog2.java
      -classpath <PATH_TO_SLOG2SDK>/src/
```

Run with java

```
java <PATH_TO_SLOG2SDK>/src/logformat/slog2/pipe/MergeSlog2
      -classpath <PATH_TO_SLOG2SDK>/src/ mpi-write-test.slog2
pvfs2-server.slog2
```

The result is shown in figure 6.4.1.

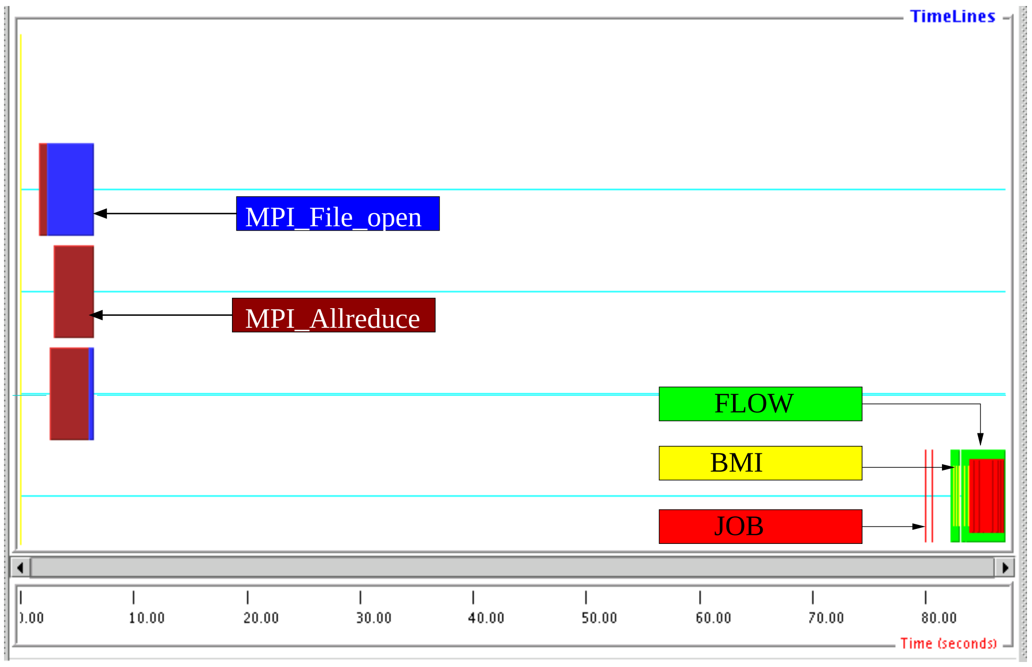


Figure 6.3: Merged SLOG2 time-line window

6.4.2 Compile and run Slog2TOCompositeSlog2

Compile with java

```
javac <PATH_TO_SLOG2SDK>/src/logformat/slog2/pipe/Slog2ToCompositeSlog2.java
      -classpath <PATH_TO_SLOG2SDK>/src/
```

Run with java

```
java <PATH_TO_SLOG2SDK>/src/logformat/slog2/pipe/Slog2ToCompositeSlog2
      -classpath <PATH_TO_SLOG2SDK>/src/ -c1 job -c2 bmi
```

```
mpi-write-test_merge.slog2
```

First possible option is to make the composite drawables using the categories. This is done be defining `-c1` <MAJOR_CATEGORY> `-c2` <MINOR_CATEGORIES>. With this option users can dissolve the overlappings in the major-category. Users can select one or more minor-categories which can be included in the major-category. In the above command we make composite drawables for JOB activities. Inside the JOB activities we include the BMI activities.

The other option is making composite drawables using an identification number defined in the primitive drawables. This can be used by defining `Slog2ToCompositeSlog2 -idorder <INPUT_FILE>`. With this option composite drawables are made for all the primitive drawables that have the same identification number.

When the primitive drawables are changed into composite drawables, it is not visible in the graphical program. To identify the composite drawables we can use the *slog2print* which has a text output.

```
slog2print mpi-write-test_composite.slog2
```

A part of the composite included SLOG2 file text output is displayed below.

```
Composite[ infobox[ TimeBBox(82.18133902549744,82.18139600753784)
Category=Category[ index=801, name=COMP_TYPJob, topo=State,
color=(255,0,0,255,true), isUsed=true, width=1, vis=true,
search=true, ratios=0.0,0.0, count=0 ]:  ]  Primitive[ infobox[
TimeBBox(82.18133902549744,82.18139600753784) Category=Category[
index=301, name=Job, topo=State, color=(255,0,0,255,true), isUsed=true,
width=1, vis=true, search=true, ratios=1.8972532,1.2359624, count=3030
]:  ]  (82.181335, 3) (82.1814, 3) ] bsize=32 Primitive[ infobox[
TimeBBox(82.18137192726135,82.18139100074768) Category=Category[ index=304,
name=BMI, topo=State, color=(255,255,0,255,true), isUsed=true, width=1,
vis=true, search=true, ratios=0.66205966,0.66122276, count=2925 ]:  ]
(82.18137, 3) (82.18139, 3) ] bsize=32 ]
```

6.4.3 Compile and run CompositeSlog2TOLineIDMap

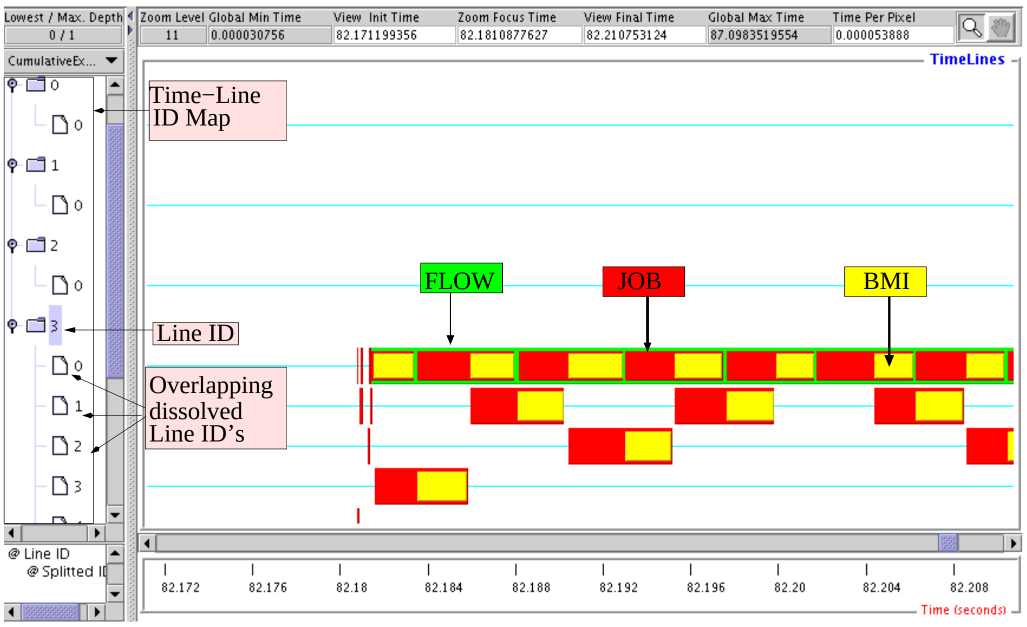
Compile with java

```
javac <PATH_TO_SLOG2SDK>/src/logformat/slog2/pipe/CompositeSlog2TOLineIDMap.java
      -classpath <PATH_TO_SLOG2SDK>/src/
```

Run with java

```
java <PATH_TO_SLOG2SDK>/src/logformat/slog2/pipe/CompositeSlog2TOLineIDMap
      -classpath <PATH_TO_SLOG2SDK>/src/ mpi-write-test_composite.slog2
```

In figure 6.4.3 we see the time-line id map generated after overlapping dissolving. We see JOB and BMI composite drawables are pushed into other lines to make them overlapping-free. The



Lowest / Max. Depth: 0 / 1
Zoom Level: 11
Global Min Time: 0.000030756
View Init Time: 82.171199356
Zoom Focus Time: 82.1810877627
View Final Time: 82.210753124
Global Max Time: 87.0983519554
Time Per Pixel: 0.000053888

Figure 6.4: SLOG2 overlapping dissolved file- time-line window

original time-line is expanded to number of different time-lines in the the time-line ID map. (Note that only a part of the whole file is displayed for better identification)

6.4.4 Compile and Run Slog2TOArrowSlog2

Compile with java

```
javac <PATH_TO_SLOG2SDK>/src/logformat/slog2/pipe/Slog2TOArrowSlog2.java
      -classpath <PATH_TO_SLOG2SDK>/src/
```

Run with java

```
java <PATH_TO_SLOG2SDK>/src/logformat/slog2/pipe/Slog2ToArrowSlog2
      -classpath <PATH_TO_SLOG2SDK>/src/ mpi-write-test_lineIDmap.slog2
```

In figure 6.4.4 we can see the the generated arrows running to and from nearby lying composite drawables. As explained in the designing chapter 4, this tool is only experimental.

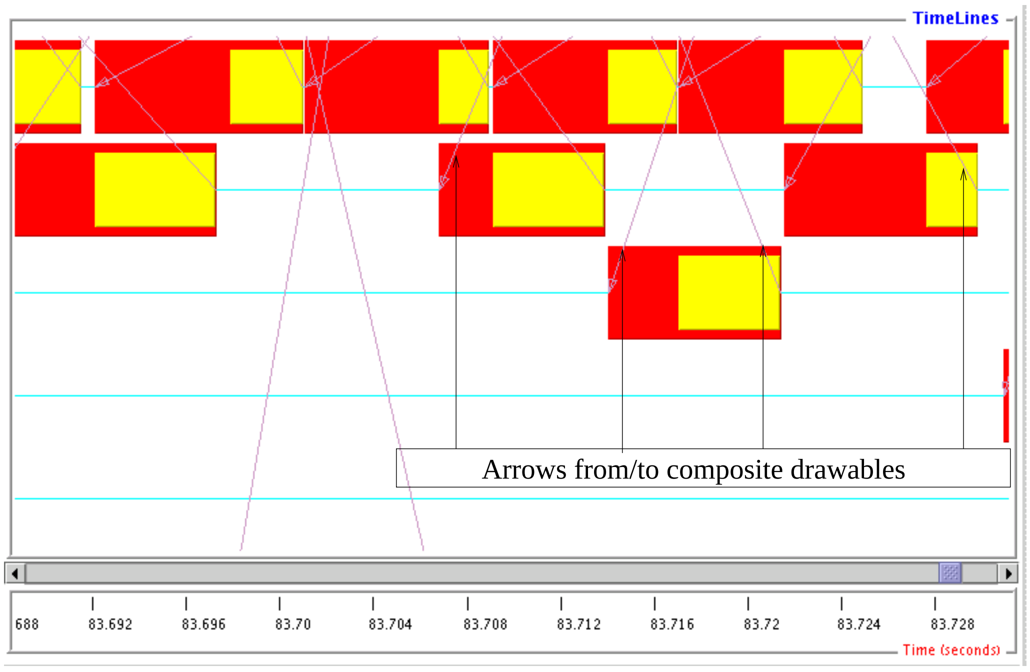


Figure 6.5: SLOG2 time-line window with generated arrows

Chapter 7

Conclusions

As we want to explicitly use the features of profiling libraries, we find that most of the general functionalities are not enough to achieve our goals. Multi-processing Environment (MPE) although well recognized and widely-used in the world can be pushed to its limits when the PVFS2 server's low level I/O activities come into play. Well defined trace file format "SLOG2" has the capability to facilitate maximum portability and easy manipulation in trace files. Although Jumpshot is good for rendering trace data and modifications can be done to facilitate different problems, most of the unused but powerful features in SLOG2 can also be used to address the same.

Conversion tools can be written to manipulate the contents of the SLOG2 file format effectively and accurately than MPE's event based file format CLOG2. Analyzing features such as search-ability and visibility in Jumpshot are helpful to identify invisible activities in the graphical program allowing more freedom in performance analysis. Above discussed programs such as Slog2ToCompositeSlog2 show that the drawable manipulation is possible when done with maximum care.

Chapter 8

Future works

This section describes most of the features that is currently needed for performance visualization in PVFS2 that are recommended by Prof.Dr.Thomas Ludwig. Possible implementation approaches from the author are explained below.

8.1 Server name

As shown in above capitals the time-line id is available only as a integer number in Jumpshot. To show a string name instead of a integer for the time-line id is currently needed by the development team in Universität Heidelberg and should be integrated to Jumpshot in near future. As this is a general limitation in Jumpshot this was referred to the Jumpshot implementers in Argonne National Laboratory[1].

8.2 Change the height of the drawable

In Jumpshot it is possible to define states (e.g a primitive drawable) that has a fixed height. The height is fixed and change it is impossible. It is better to give more information by defining the height of the drawable in some practical situations. Hereby defining the height of a drawable can be used to represent more information.

There can be two alternatives to solve this problem.

First one is to add extra information to a state drawable that can be identified by Jumphshot and accordingly displayed in the graphical canvas. This will be less time intensive to develop and will need only changes in Jumpshot time-line window. But the changes will only speak this particular need and future enhancements will be complicated.

Second idea is to define a new state, which has a variable height or the height as percentage that can be integrated into SLOG2 drawable format and also to Jumpshot.

[1]currently the main developer of Jumpshot is Anthony Chan and he can be contacted with chan@mcs.anl.gov

This approach will by more time intensive than the first because SLOG2 API should also be changed accordingly to accommodate the new defined state. But this will be a more meaningful general feature that each trace implementer can use.[2]

8.3 Change the color depth of a category

Currently Jumpshot identifies all the drawables in a particular category with a color that is defined in the SLOG2 file. If this feature can be modified to display more colors for a particular category, it will be more helpful in analyzation processes.

As an example take the BMI activity in PVFS2. After profiling how many bytes were sent during each BMI activity, this information can be used to visualize the BMI activities which lasted longer in a different color than the ones which had a shorter duration.

Author suggests to integrate this feature in the SLOG2 drawable as a separate information field. Jumpshot should be modified to identify this as separate information and give the color accordingly to the drawables.

Another possible approach is to modify the SLOG2 to group the categories into sub categories.

As an example BMI activity category can be divided into LONG_LASTING_BMI and SHORT_LASTING_BMI. This can be done by adding an identifying algorithm in SLOG2 conversion tool. This algorithm can insert the short lasting BMI activities into BMI_SHORT and the long lasting BMI activities to BMI_LONG. Original category of BMI is now divided into BMI_LONG and BMI_SHORT. Thereby defining two new categories and deleting the old one. No changes to Jumpshot is needed. This is generally applicable if the number of categories is small. But if there is a large number of categories this approach is not suitable.

[2]this was referred to Anthony Chan

Chapter 9

Possible future enhancements

9.1 Automatic color changing

After adding the modifications discussed in the section 8, Jumpshot will facilitate more colors to improve optical identification of drawables. Now the trace implementer will need to specify colors more carefully, because of the number of colors will large and he has to take care not to use the same color twice that will make the visualization useless. Defining user-friendly colors is desirable and can be done by trace implementer but this can also be given to the graphical program.

A possible solution for this is to add pre-defined color schemes to Jumpshot that the end-user can select. This will also help to overcome the problem of different color schemes in different monitors. If the colors defined in SLOG2 are not acceptable for a particular user he can switch to a different color scheme which will render a better picture for him. Therefore it is recommended to give the possibility of adding color schemes that can be changed by the user. Author suggests that this feature can be integrated into time-line window because redrawing (or refreshing) facility already exists in the time-line window.

9.2 Integrate searching facility for activities

Currently users of Jumpshot are provided with the options of visibility and search-ability for single primitive drawables. Users see all the categories in the search window and they can click the category for their visibility and search-ability. (see more on Jumpshot user's guide [3] about this feature) Clicking over a large number of buttons can be time consuming and confusing if there is a large number of different categories and the user has to filter some of them. A searching facility can help to solve this problem if the categories are named accordingly. As an example we can take our approach of profiling MPI and PVFS2. If we name all the MPI activities as MPI_*XXX* and PVFS2 activities as PVFS2_*XXX* then a search phrase like MPI will filter all the MPI related categories. Search functionality can be integrated in legend window by enhancing its current capabilities. (Don't confuse the search-ability with the search option. Search-ability option in legend window make single drawables searchable within the time-line window)

9.3 Two time-line windows at the same time

Time-line windows depend on the time-line id map. Jumpshot lets the user choose a time-line id map in main window and redraw a new time-line window accordingly. It makes sense to assume that this approach was taken to save main memory(RAM) because two or more time-line windows for a large SLOG2 file will take a long time to be drawn in the monitor. But if there are enough resources after adding different line id maps, a second time-line window will be helpful for better analyzation purposes. To illustrate this in an example, take a MPI Program that is visualized in Jumpshot. Program has run on more than one node. We want to see how it looks like when we exchange the node numbers referring to the the original time-line window at the same time. Currently this is not possible as the first time-line window is automatically closed when the second one is started. One may argue that starting Jumpshot with the same trace file twice will avoid this necessity. Therefore author suggests that this function can be added as an *additional* feature in the main window.

Chapter 10

PVFS2 installation with mpe

Edited by Julian Kunkel and Stephan Krempel. Translated by Dulip Withanage.

Both PVFS2 and MPICH2 can be built, with the help of "VPATH" principle. That means, simply not to use the the directory tree of the source code for compiling and configuring. This help to resolve possible problems with the revision control system. This approach also makes possible to have different configurations from the same source code(which is a necessity in having 2 parallelly installed MPICH2 versions)

Assumptions:

```
$HOME/local/mpich2-src          //  MPICH2 Source Tree
$HOME/local/mpichPLAIN          //  Destination for MPICH2 without PVFS2
$HOME/local/mpichPLAIN.build    //  Build-directory for MPICH2 without PVFS2
$HOME/local                     //  Destination for MPICH2 with PVFS2
$HOME/local/mpich2.build        //  Build-directory for MPICH2 with PVFS2
$HOME/local/pvfs2-src           //  PVFS2 Source Tree
$HOME/local/pvfs2.build         //  Build-directory for PVFS2
master1,node01,node02,node03    //  PVFS2 I/O Servers
master1                         //  Meta- Data Server
```

Steps on how to install PVFS2 with changes

MPICH2 without PVFS2

```
mdkir $HOME/local/mpichPLAIN.build
cd $HOME/local/mpichPLAIN.build
$HOME/local/mpich2-src/configure - -with-mpe - -prefix=$HOME/local/mpichPLAIN
make ; make install
```

PVFS2 with MPE and mpiexec

```
mkdir $HOME/local/pvfs2.build
cd $HOME/local/pvfs2.build
```

```
$HOME/local/pvfs2-src/configure - -with-mpiexec=$HOME/trace/
mpichPLAIN - -with-mpe=$HOME/trace/mpichPLAIN - -prefix=$HOME/trace
```

Before make, change the permissions of the shell scripts of the following directory, to avoid "change of permission, denied"

```
chmod 755 $HOME/trace/pvfs2-src/maint/*.sh
make ; make install
```

MPICH2 with PVFS2

```
cd $HOME/local
mkdir mpich2.build
cd $HOME/local/mpich2.build
export CFLAGS="-I$HOME/trace/include"
export LDFLAGS="-L$HOME/trace/lib"
export LIBS="-lpvfs2 -lpthread"
$HOME/local/mpich2-src/configure - -enable-romio - -with-file-system=ufs+nfs+pvfs2
- -prefix=$HOME/trace - -with-mpe
make ; make install
```

Start the MPI environment

Be careful!
Cluster can be already configured to take the MPI in it's PATH. Then you have to change the PATH to use self-compiled version. This can be done as follows.

`export PATH=$PATH/local/bin/:$PATH/local/sbin:$PATH` Set MPD password:

```
cd $HOME

echo "secretword=blubb" >  /.mpd.conf
```

Define MPD hosts

```
echo -e  master1\nnode01\nnode02\nnode03" > mpd.hosts
```

Start the MPD's

```
cd $HOME
mpdboot - -totalnum=4 - -rsh=rsh - -file=mpd.hosts
```

Test the MPD's

```
mpdtrace
```

Result should be like this:

```
master1
node03
```

```
node02
node01
```

Configure PVFS2

```
cd $HOME/local/pvfs2-src/
```

Please change the variables for port and log file `pvfs2-kunkel` and `pvfs2-kunkel.log` accordingly to avoid two users using the same port and same storage directory.

```
pvfs2-genconfig - -protocol tcp - -tcpport 3449 - -ioservers
master1,node01,node02,node03 - -metaservers master1 fs.conf server.conf
- -storage /tmp/pvfs2-kunkel - -logfile /tmp/pvfs2-kunkel.log
```

Set the storage space:

```
cd $HOME
mpiexec -np 4 pvfs2-server -d -f fs.conf server.conf
```

Start PVFS2:

```
cd $HOME
mpiexec -np 4 pvfs2-server -d fs.conf server.conf
```

Result:

After last command the terminal should hang on . With the following Command program can be started in background.

```
 mpiexec -np 4 pvfs2-server -d fs.conf server.conf&
```

Set up the pvfs2tab and define the port for PVFS2:

```
 cd $HOME
echo "tcp://localhost:3449/pvfs2-fs /pvfs2 pvfs2 default,noauto 0 0" >
pvfs2tab
```

Set the `$PVFS2TAB_FILE` (Add it to the profile as default, otherwise you should define every time you want to log in)

```
export PVFS2TAB_FILE=$HOME/pvfs2tab
```

Test PVFS2 `export PVFS2TAB_FILE=$HOME/pvfs2tab`

```
pvfs2-ping -m /pvfs2
```

Compile a test program and test

```
cd $HOME/local/pvfs2-src/test/client/mpi-io/
mpicc mpi-io-test.c -o mpi-io-test -llmpe -lmpe
pvfs2-set-eventmask -m /pvfs2 -a 0xFFFF -o 0xFFFF;
mpiexec -np 4 ./mpi-io-test -f pvfs2://pvfs2/test;
pvfs2-set-eventmask -m /pvfs2 -a 0 -o 0;
```

Comments:

Remove the /tmp/pvfs2-server.clog2, as others may not be able to write their files if you don't remove it. You can remove it using

```
mv /tmp/pvfs2-server.clog2 $HOME
```

Bibliography

[1] Ewing Lusk Anthony Chan and William Gropp. From Trace Generation to Visualization A Performance Framework for Distributed Parallel Systems.

[2] Ewing Lusk Anthony Chan and William Gropp. User's Guide for MPE: Extensions for MPI Programs. `http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpeman/mpeman.htm`, 2002.

[3] Rusty Lusk Anthony Chan, David Ashton. Jumpshot 4 Users Guide. `http://www-unix.mcs.anl.gov/perfvis/software/viewers/jumpshot-4/`, 2005.

[4] Anthony Chan, William Gropp, and Ewing Lusk. Scalable Log Files for Parallel Program Trace Data. `ftp://ftp.mcs.anl.gov/pub/mpi/slog2/slog2-draft.pdf`, 2000.

[5] Thomas Ludwig Julian Kunkel and Hipolito Vasquez. Weit verteilt - Dateisystem für parallele Systeme: PVFS, Version 2 . 2004.

[6] Robert ross Rob Latham, Neil Miller and Phil Carns. A Next-Generation Parallel File System for Linux Clusters. `http://www.pvfs.org/pvfs2/files/linuxworld-JAN2004-PVFS2.ps`, 2004.

[7] PVFS2 Development Team. Parallel Virtual File System, Version 2. `http://www.pvfs.org/pvfs2/pvfs2-guide.html`, september 2003.

[8] Vampir. Performance Analyzer. `http://www.hlrs.de/organization/par/services/tools/performance/vampir.html`.