Scalable Log Files for Parallel Program Trace Data
DRAFT

Anthony Chan, William Gropp, and Ewing Lusk

May 16, 2001

1 Introduction

A powerful technique for understanding the behavior and performance of parallel programs is the visualization of trace files (also called log files) collected during the execution of the program. A trace file contains several basic elements. Typically, these are generated during the execution of a program by very short code sequences (so as to minimize the perturbation of the execution caused by the tracing [7]), and are either written to disk (buffered, of course) or to memory as they are generated. Tracefiles typically contain sequences of *events*; an event has a timestamp and some data. A collection of events for a single process, thread, or processor is sometimes called a *timeline*.

Such *post mortem* analysis based on trace files has been an important tool [5, **?**, **?**, 9, 4, 3, **?**, 6, 1, 8, 10, **?**, **?**] for performance analysis. Many of these tools display a trace file as a GANTT chart, with the x-axis representing time and the y-axis process or thread number. However, as parallel programs use increasing numbers of processes or threads and run for longer times, the amount of data that is collected into a trace file can become extremely large, exceeding hundreds of megabytes.

One reaction to the problem of displaying this amount of data has been to summarize the data; for example, displaying total event counts and distributions of times within each state. Unfortunately, it is sometimes necessary to examine the detailed behavior of a program to understand it. Indeed, over the years, we have continued to find most useful the ability to examine small time intervals in considerable detail. The GANNT chart approach, in which the "state" of a process is represented by a colored bar extending over a time interval and can be compared visually with the states of other processes at the same time, has proven effective. Many tools (see [10], for example) augment this view with further detail such as arrows to show messages and popup windows to display detail data on process states or messages. However, the basic problem of *scalability* arises even when the simplest form of GANNT chart is being displayed, and the issues discussed in this paper can be thought of in this context.

Therefore we state the general problem we address as follows. We assume that a parallel program produces as it runs a large volume of data on program behavior, including states of processes varying over time. We make no assumptions about the lengths (in time) of those states. We wish to design a system that will support the graphical display of this data in a scalable way. Scalable display means that the CPU time and memory requirements for display of some time interval about a particular time depend on the number of graphical objects to be displayed and not on the total amount of program data nor on the particular time chosen.

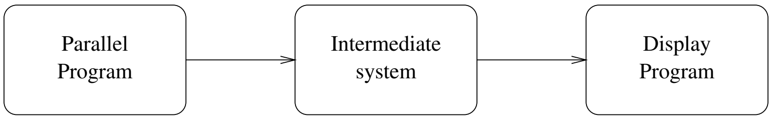

Figure 1: High-level view

In this paper we describe an architecture for the "intermediate system" that links the parallel program producing the data with the graphical display program (See Figure 1). By defining the appropriate API's, we will keep the architecture independent of either the mechanism for collecting data or the display program. We do anticipate being able to link specific systems, however. On the trace data production side, we expect to be able to utilize the MPI profiling interface or the UTE files produced by IBM's AIX-based system. On the display side, we expect to support a system like Jumpshot [10]. A rough approximation of its appearance (minus the colors) is shown in Figure 2.

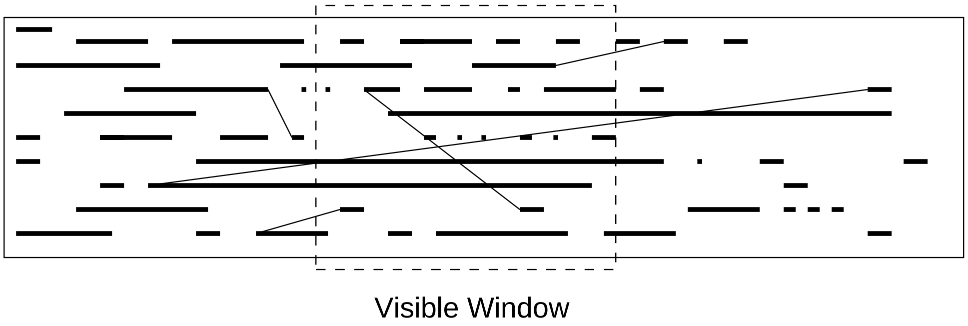

Figure 2: Example of a log file display. Only the data within the dashed box is displayed; the visualization program must render all data as if the entire data is displayed, clipped to the dashed box. This includes the rectangles that enter and exit the dashed box, and the lines connecting the rectangles. Note that in practice, the time interval shown by the visible window may be only 1% or less of the entire trace file.

Jumpshot works by providing an end user first with a *Preview* interface. This is a mechanism for displaying data from the entire run in a way that trades away details of display for the ability to examine the entire run and select a time (instant) of interest. Once the user selects a time, jumpshot chooses an interval of time containing that time and provides a detailed view of that interval. Scalability is achieved by allowing jumpshot to choose the width of the time interval and thus the number of graphical objects to be displayed. We describe Jumpshot here only to make concrete the requirements of the intermediate system. In particular, the system needs a way to manage summary data that will be used in the Preview. More importantly, it needs to be able to rapidly provide (without reading large amounts of the file) to the Display Program the graphical objects in a given time interval. When the objects extend beyond the interval being displayed, that portion of them that intersects the interval must be displayed. The main idea of this paper (the "bounding box" approach) is our solution to this problem.

One possible architecture for the intermediate system would be a *file* with a specified format for supporting both the necessary indexing scheme for directly accessing a contiguous section of the file based on time and some mechanism for accessing objects that extend over long periods of time. We originally took this approach, calling the file format SLOG (for scalable log file), now called SLOG-1. The specification of API's for reading and writing this file is a better approach, since it frees the file format itself for independent optimization (such as compression). This approach was used in the later stages of SLOG-1 (see Figure 3. Different data sources were accomodated by having completely independent programs that wrote SLOG-1 files.

Figure 3: SLOG-1

In SLOG-2 we define also an input API for the program (the "SLOG Program") that writes the SLOG file (if it exists), *and* we allow more flexibility between the SLOG Program and the Display Program. In particular we will describe how, instead of single complete SLOG file, the SLOG program could write instead an SLOG Annotation File, which can be used in conjunction with the original data to provide the necessary scalability functionality to the Display Program. The use of annotations allows this approach to be retrofitted onto any existing trace-file collection system, and allows the different annotations to be generated (emphasizing different features) for a single collection of trace files. For example, one set of annotations might organize the display around processes and another, quite different one, may use threads or even user-tasks. In addition, by using annotations, it is easy to provide a way for the user to add and record annotations about a particular run. The key ideas of this paper apply to both complete SLOG files and SLOG Annotation files. We focus on the API's used to access data, rather than file formats. The complete model is given in Section 2. Section 3 describes the scalable implementation of the annotations, including a one-pass algorithm for their creation. A one-pass algorithm is essential since we assume that the size of the trace files is extremely large.

Section 4 describes some enhancements that further improve the SLOG file, along with an analysis of the design choices. Finally, Section 5 shows the results of some experiments with our implementation of SLOG.

2 Terms

A system whose end result is the graphical display of performance data may consist of a number of different files and programs. In addition, well-defined interfaces between various components allow different configuration of files and programs to produce the same end result. In Figure 4 we show our view of these components. It is important to realize that not all these components need be present in a given instantiation of this very general architecture.

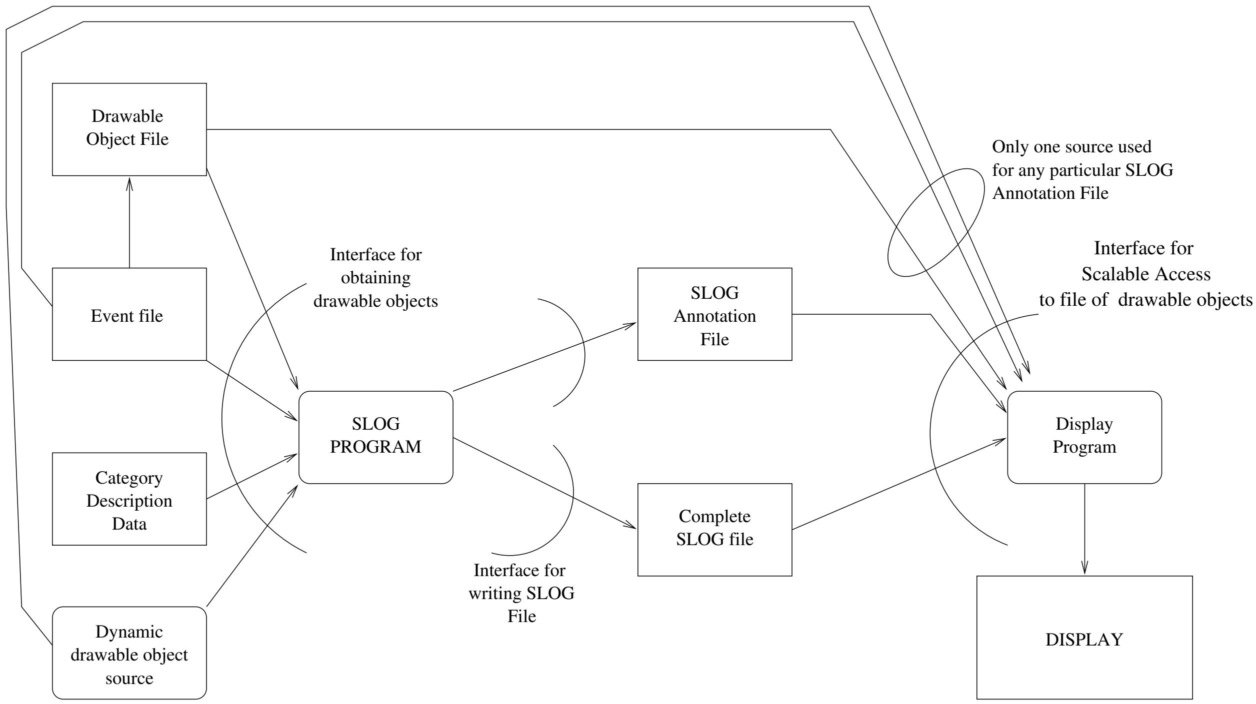

Figure 4: Components of an SLOG-based system

The central component is the **SLOG Program**, whose function it is to contribute scalability to the system. It is written in terms of two API's. Its input comes from the **Drawable Object API**. The fundamental function in this API is the `get_next_drawable_object` function. It can be implemented in a variety of ways:

- There may be a pre-existing file whose records consists of drawable objects. We call this the **Drawable Object File**, or DOF. In this case, the `get_next_drawable_object` function just returns the next record.

- There may be no Drawable Object File, but rather a more primitive file of timestamped events, which we will call an **Event File**. In this case the get_next_drawable_object function may read a record from the event file and if it completes a drawable object, return that drawable object, otherwise stack the event and read the next event, continuing until it has assembled a complete drawable object.

- The get_next_drawable_object function may alternatively be reading from a stream that is being written by another program. That program may be writing either drawable objects or events or both onto its end of the stream. In the case of events that do not comprise drawable objects, again a stacking and matching algorithm comes into play.

Drawable objects are objects destined to be represented as (virtual) single graphical objects in the final display. Examples would be program states to be drawn as rectangles, message transmissions to be drawn as arrows, or collective operations to be drawn as polygons. The important property that they all share is that they have some bounded extent in time. The word "virtual" is used because the display may be only able to show part of one of these objects at a time. One of the central problems of scalable display is to be able to cope with objects that have a comparatively long extent in time, longer than the time interval that can be viewed at once in the display.

In all three of the above cases, an important feature is that the matching of event records to form drawable objects (e.g., the matching of a start_send event with an end_send event to form a drawable MPI_Send state), which requires a stack of events, is *not* done in the **SLOG Program** but rather in one of the several possible implementations of the SLOG Input API.

The output of the **SLOG Program** can be either a **complete SLOG file**, containing all the data needed by the display program, or it can be an **SLOG annotation file**, which is designed to be used in conjunction with a drawable object file to produce the display.

Finally the **Display Program**, using an API for reading the data necessary to graphically represent a particular time interval, reads and renders subsets of the drawable objects around a time selected by the user. We term the whole system *scalable* if the time and memory requirements necessary to display some time interval (whose duration is chosen by the display program, not the user) around a given time is approximately independent of the total number of events or graphical objects that are produced by the original sources. In a scalable system, the time and memory requirements for display of a time interval is proportional to the number of visible drawable objects whose duration overlaps that time interval, and is independent of the total number of drawable objects or the location of the time interval within the total time interval of the run. Scalability is achieved by letting the user pick the time (instant in time) but not the duration of the time interval around it.

In order to display the drawable objects, the Display Program needs to consider several types of information:

- The information specific to each object, consisting of the start and end time of the object (used by the SLOG program to provide scalable access to the data), along with the other graphical coordinates needed to draw the object. In addition, a drawable object may have a byte string that contains information needed to provide other (usually non-graphical) information about the object, such as the location in the source code where the object was added to the trace file or the parameters of a routine call. Since many different drawables will be rendered in the same way, a drawable specifies which *category* it belongs to.

- Information common to all objects of the same category, such as the shape, color, label, the format of the object-specific data, and a specification of how to display the object-specific data. (E.g., a `printf` string). This information will be used for the Legend and the Popups in the Display Program. To allow the user to invoke a variety of actions when a particular drawable object is selected, each category can specify a collection of *methods*.

- Methods simply describe the action to take with the byte string attached to a drawable. The simplest method displays the byte string as text. More sophisticated methods may display the parameters to a routine, complete with labels, popup a source code browser highlighting the location in the source code that added the drawable to the trace file, or even produce an alternative format for the graphical display (e.g., the bebit problem).

- The graphical coordinates themselves come in two types. The first is the time value, of which the start and end time are two examples. Since the typical display of a trace file is two-dimensional, with the time axis going left to right across the display, we will call the other coordinate type the *y coordinate*. Further, the y coordinate is discrete; a typical display may have only 16 to 64 distinct y values. In addition, the y value may be a small positive integer, such as the number of the process, starting from zero, or it may be a more complex value, such as a tuple that includes the node id, cpu id, process id, and thread id (still specified as an integer). In order to handle this more complex case, we allow a level of indirection in the y coordinate values. Each drawable object specifies y coordinates as integer values and in addition specifies how to map these values to a small positive integer. A trace file may define several such maps, such as one for MPI process rank, one for node number, and one for each thread. This is accomplished with a *coordinate map*. If no coordinate map is provided, the obvious identity map is used (this is appropriate for trace files that use MPI process rank as the y coordinate).

For scalability, it is important that the Display Program have a *preview* function that aids the user in picking a time (not a time interval) to look at. The data (for example, statistics on the distribution of objects in the file) necessary to provide the preview are in the Complete SLOG File or the SLOG Annotation File.

2.1 Examples

We have experience with two approximate instantiations of the architecture described here, although neither one is a perfect match. Each of them uses Jumpshot [10] as its display program. We refer to the version of April, 2000 as SLOG-1.

2.1.1 SLOG-1 with UTE Files

We have published an interface for writing complete SLOG-1 files that are read by Jumpshot. IBM has written a program that uses this interface, although it is not currently architected like the SLOG Program described above. It contains the stacking algorithm to read UTE files, whose records are not the graphical objects to be displayed by Jumpshot. The event files that are processed into UTE files are produced by the AIX trace mechanism.

2.1.2 SLOG-1 with CLOG

We have also written a profiling library that uses the MPI profiling interface to implement dynamic logging of events that occur in MPI programs. At the time of `MPI_Finalize`, logs located on various processes are merged into a stream, and records are written out to an SLOG File using the SLOG-1 API. Thus code inside the profiling version of `MPI_Finalize` plays the role of the SLOG Program, although it does not yet use a `get_next_drawable_object` interface, but rather does the stacking and matching itself in open code.

2.2 Term Definitions

At some risk of redundancy, we list here the components described narratively above. Terms in bold face refer to other items in this glossary.

Display Program The end result on the screen. We expect it to support at least:

- A **Preview** representing the whole run, which the user can use to help choose a time value of interest.
- A **Timeline View** or **GANNT Chart** display of the graphical objects or portions of them that fall within a time interval containing the chosen time. The overarching goal of the project is to make this display scalable in the sense defined above.
- A **Legend** that labels graphical objects in the Display.
- An optional set of **Popups**, containing text data about the object.
- A display of **Drawable Objects**, each of which is a single graphical object. It has an extent in time given by its starting time and duration. It may also have attached to it a byte sting representing data associated with this instance of the object. It has an object category that connects it to the **Category Description Data** and specifies what type of graphical object it is to be rendered as, along with sufficient data to render the object. It has at least one y-coordinate that is used to place the drawable object on the correct timeline(s) (the name y-coordinate is used because it matches the usual display (Figure 1). The **SLOG Input API** will enable `get_next_drawable_object` and related routines to access all information in the Drawable Object and interpret it. Its is important to realize that the Drawable Object is a virtual object defined by the **SLOG Input API** and there may or may not be a **Drawable Object File**.

  The most common drawable object is a *state*, defined as a duration (or interval) in time. States are usually drawn as rectangles. A more complex object might be a sequence of separate rectangles on a single timeline representing periods where a thread is running during a state.

The **display program** uses the **SLOG Input API** to read from the SLOG file.

SLOG Program A program whose function it is to add scalability to the system. It uses a well-defined API to obtain a set of drawable objects and another to write out either a **complete SLOG File** or an **SLOG Annotation File**.

Complete SLOG File A single file containing all information necessary as input for the **Display Program**.

SLOG Annotation File A file containing just the "scalability" part of the **Drawable Object** data. In this configuration, the data is left in either a **Drawable Object File** or the **Event File**. The **Display Program** accesses the data through the **SLOG Input API**.

The SLOG Program uses two APIs to read and write data:

DO Input API (Drawable Object Input API) The programming interface for delivering **Drawable Objects** to the **SLOG Program**.

SLOG Output API The programming interface for storing drawable objects and other data into an **SLOG File** or into an **SLOG Annotation File**.

SLOG Program input consists of drawable objects and related information. Depending on the implementation of the **SLOG Input API**, the sources may be

Drawable Object File A file whose records are drawable objects. Its existence is optional.

Event File A file (optional) of lower level data (not assembled into **Drawable Objects**) that may be used directly by the **SLOG Input API** or may be processed by a separate program into a **Drawable Object File**.

Dynamic Drawable Object Source A program that emits a stream of **Drawable Objects** that is consumed by the **SLOG Program**. It may obviate the need for any input files to exist for the **SLOG Program**.

In the typical case, the drawable objects belong to a relatively small number of categories, such as `write`, `MPI_Send`, or `User-routine`. These are described by

Category Description Data This is another optional file. Precisely how this data is delivered to the **SLOG Program** is defined by the **SLOG Input API**. The data is that common to all drawable objects of the same category, such as shape, color, label, data format of instance-specific data, and a specifier for how that data is to be displayed.

2.3 SLOG-2 Plans

An important part of SLOG-2 is a precise description of the all the various APIs; see Appendix A, B, and C. These APIs will encourage multiple instantiations of this architecture. Besides defining the architecture more clearly, there will be two completely new features:

- The structure of an SLOG File will be completely different from that of SLOG-1. In order to deal with graphical objects that span "long" periods of time, we will use what we call a "bounding box" approach, described later in this paper. Pseudo records, the mechanism used for this purpose in SLOG-1, will eliminated. Associations will be replaced by a more general notion of drawable object.

- We will explore the idea of having the SLOG Program emit an SLOG Annotation File to be used in conjunction with an original Event File or Drawable Object File, thus saving both time and disk space. The fundamental idea of bounding boxes is independent of whether the SLOG Program writes a complete SLOG file or an SLOG Annotation File. We hope to be able thus to add scalability to existing Drawable Object Files such as those produced by other tools such as Vampir or PICL, as well as unprocessed UTE files.

There will be several advantages to converting to SLOG-2:

- The architecture of the system will be cleaner, with the SLOG program's role simplified to adding scalability, while the assembly of drawable objects will be moved to the other side of the SLOG Input API, which in turn can have multiple implementations.

- The elimination of pseudo records saves space, complexity, and time.

- Only one pass through the Drawable Objects will be necessary, saving time when the SLOG File is created. It will be easy to do this if the SLOG Input API has a routine to get the total run time, but we describe later how it will be possible even if the SLOG file is being created during the run.

- SLOG Annotation Files will allow one to use SLOG and Jumpshot with existing file formats, such as UTE, without copying their data into a Complete SLOG File.

Note: In SLOG-2, responsibility for determining which graphical objects will be displayed is moved out of the SLOG Program and into the implementation of the SLOG Input API. How to render them is moved into the Display Program. This is different from the SLOG-1 viewpoint that assembly of drawable objects is done jointly by the SLOG Program and the Display Program. This clarifies and simplifies the SLOG Program and provides more flexibility in the interfaces, but may require changes to the ways we currently do things in the IBM collaboration.

An illustrative example is given by "bebit" processing. Where are little intervals combined into big intervals? Do the various types of fragments (different bebit values) belong to different categories or the same category? They should because they have the same color and label; they shouldn't because they contain different data. In the SLOG-2 approach, there are two alternatives:

- If we want both big intervals and fragments as drawable objects, then they are of different categories, and both go into the SLOG File. Jumpshot displays either the big ones or all the types of little ones or both.

- We define a new single drawable object, consisting of lots of rectangles. (So the geometric data is variable and large.) Jumpshot knows two different ways to render this object.

How to deal with cpu/node/thread ids is left entirely to Jumpshot.

We need four API's altogether:

- One to read a tracefile. Implementations will be specialized for reading various tracefiles (UTE, CLOG, ALOG, etc.)
- One to write an SLOG2 file. The implementation of this will create the tree.
- One to read an SLOG2 file. Implementation in C, for testing.
- Another one to read an SLOG2 file. Implementation in Java, for Jumpshot.

The DO API, described in Appendix B, is designed to *not* impose a particular structure for the drawable objects on the user. Instead, each drawable object is described by some text (stored at `text_base + text_pos`), coordinates (stored at `coord_base + coord_pos`), a category, and a time range. The API makes it easy to read these into either a single struct representing a drawable or into separate arrays for coordinate and character data. The routine `SLOG_Peek_next_drawable` makes it possible to preallocate space for the next drawable.

The SLOG Output API is described in Appendix C and the SLOG Input API is described in Appendix D.

Records are viewed as graphical objects. The object types are rectangle, line, arrow, and polygon. In addition, there is a special object type called "bebit rectangle" that describes a collection of rectangles that can be displayed either as a single rectangle or as individual rectangles. Each record may have additional data that can be displayed by calling the routine to create a text description of the record.

Typically, a visualization program such as Jumpshot [10] will access a range of records, so any implementation of these methods should assume that most read operations access successive records.

SLOG-1 had the concept of related records. While this helps describe some relationships, managing record relationships adds significant complexity to the system. In SLOG-2, there are no related records; instead, there are complex drawable objects. In other words, what in SLOG-1 was several related drawable objects is in SLOG-2 a single but more complex drawable object. This provides a cleaner, more modular interface.

2.4 Popup Methods

The popup methods describe how the drawing program should interpret the byte string of data that each drawable object may have. In the simplest case, the byte string contains printable characters and the method pops up a simple text box containing this data. A more complex method may know how to format the data with labels or access a source code browser.

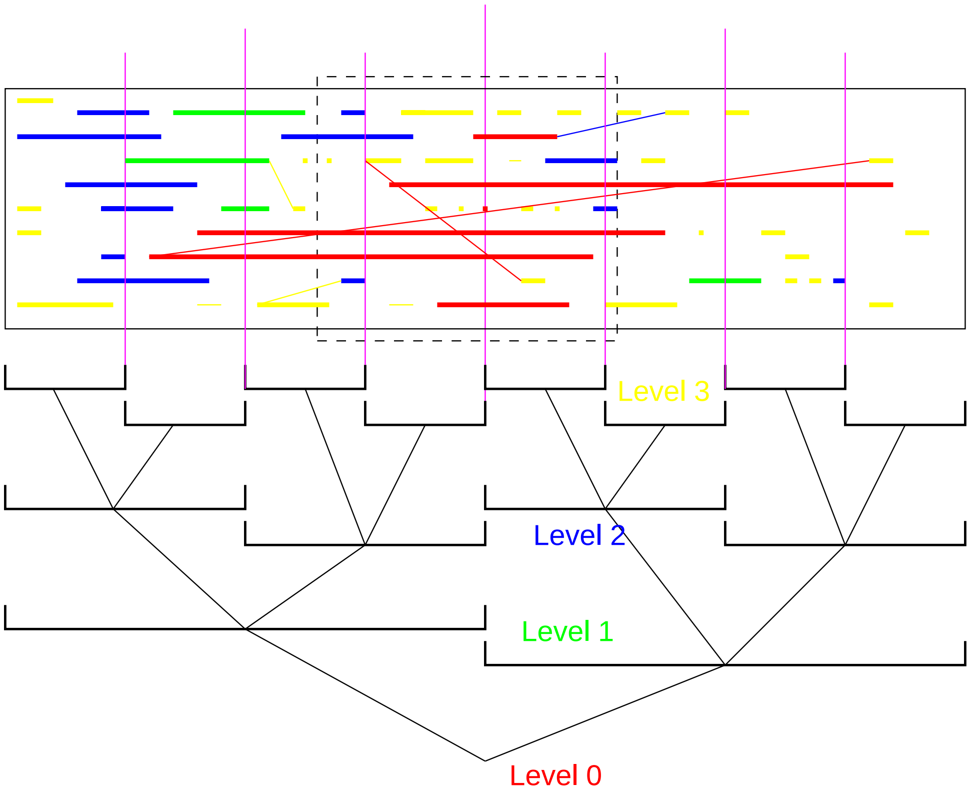

Figure 5: Bounding boxes for the log file in Figure 2. The figure shows the intervals defined as bounding boxes as horizontal brackets, along with the hierarchy of bounding boxes. Colors indicate the assignement of graphical objects to bounding box levels.

3 SLOG Annotations

The goal in SLOG is to enable the display of graphical objects described by one or more trace files. One way to look at this is that we wish to display a small region of a much larger picture, as shown in Figure 2.

A common way to organize data of this kind for graphical display is to define bounding boxes. This approach provides a simple and efficient way to access only the data necessary to draw the region that overlaps the bounding boxes. This approach is shown schematically in Figure 5.

Perhaps the best way to describe the SLOG annotation for the trace data is to describe the algorithm for computing the tree shown in Figure 5. For simplicity, we will assume that the trace records cover a time interval $[0, T]$ and that the data in the original trace file consists of rectangles sorted by end time. That is, each rectangle is described by a thread number (y coordinate in the visualization) and a time interval $[t_s, t_e]$ (x coordinate in the visualization). The rectangles are sorted by end time, t_e.

The SLOG annotations simply form a binary tree, defined recursively as follows. The root node represents the time interval $[0, T]$. For any node, representing the time interval $[t_1, t_2]$, there are two children representing the time intervals $[t_1, \frac{1}{2}(t_1 + t_2)]$ and $[\frac{1}{2}(t_1 + t_2), t_2]$. A node is a leaf of the tree if the size of the time interval is less than or equal to ΔT_{min}. (We describe a number of generalizations to this definition in Section 4.) Records (rectangles) are placed into the smallest (in time interval) containing node.

To display any graphical objects around any chosen time, only a subset of the nodes of the tree must be read. Specifically, to display a time t, only the nodes of the tree whose time intervals intersect with this time must be read and displayed (if t is near either end of an interval, we may choose to read the adjacent interval in order to provide a more informative display).

Under the assumption that records are uniformly distributed throughout the trace file, the value of ΔT_{min} can be computed as follows. Assume that, for efficient display, the data read should be limited to n_{max}. Further, assume that the size of the file is n_f, the display will show only a single time interval at any time. Then

$$\Delta T_{min} = \frac{1}{2} T 2^{\left\lfloor \frac{\ln \frac{n_f}{n_{max}}}{\ln 2} \right\rfloor}$$

(The extra factor of two allows the display program to read two leaf nodes, since any time interval of duration ΔT_{min} will intersect with at most 2 leaf nodes.)

In the annotation model, the leaves of the tree are desribed by a start and end position within a trace file; there may be at most one file for each timeline (this requirement can be relaxed by defining a virtual file; after all, the only access to the trace "file" is through the API). In this case, the SLOG file contains only the non-leaf nodes of the tree. As we show in Section 4.5, the resulting SLOG file can be quite small.

3.1 Single Pass Creation of SLOG File

Assume that we have a trace file containing records, sorted by end-time. Recall that a record represents a single drawable object of with a known start and end time; the simplest such object is a state, often drawn as a rectangle. We wish to create an SLOG annotation for this file in a single pass. We will further assume that most records will fit in the leaves (i.e., their duration is less than ΔT_{min}). Our algorithm creates a postorder representation of the tree; that is, when moving sequentially through the SLOG annotation, the tree nodes are visited in postorder (children before parents).

Let the root of the tree have level 0 and define the level of the children of a node a one greater than the level of that node. The tree has levels 0 through L. For each level ℓ, let there be a list of records R_ℓ, initially empty. In addition, for each level ℓ, there is a time interval ΔT_ℓ that specifies the time interval for the current tree node on level ℓ. Finally, we keep track of the location of the data in each node of the tree within the file in a *directory*; this directory has $2^L - 1$ entries, and is relatively small. The directory should be in preorder to simplify searches. In order to simplify the description, each node on level ℓ is covers a time interval of length $2^{-\ell} T$. Section 4, point 5 discusses an alternative that limits the amount of memory used by each leaf node.

The algorithm to write an SLOG file is shown in Figure 6.

We could also write the directory at the beginning of the file, since we know how long it is, but for reasons discussed below, we put it at the end. Normally, the lists R_ℓ can be maintained in memory, with the for $\ell = 0, 1, \ldots, L$ do {

    Set R_ℓ to empty
    Set ΔT_ℓ to $[0, 2^{-\ell}T]$

}

Open trace file

while not done {

    read the next record r.
    accumulate any statistics or coordinate mapping data on r
    for $\ell = L, L-1, \ldots, 0$ do {
        if the end time of r exceeds the end time of ΔT_ℓ, then {
            Write R_ℓ out. Record the location of R_ℓ in the file in directory D. Set R_ℓ to empty.
            Set ΔT_ℓ to the next time interval (add $2^{-\ell}T$ to the interval).
        }
        if the time extent of r is contained within ΔT_ℓ, then {
            Add r to R_ℓ
            break from for loop
        }
    }

}

for $\ell = L, L-1, \ldots, 0$ do {

    Write out list R_ℓ. Record the location of R_ℓ in the file in directory D.

}

Write out any gathered statistics or other summary data

Write out the directory D.

Write out the offset (from the end of the file) to summary data

Write $2^L - 1$ (the number of directory entries) as an integer

Figure 6: One-pass algorithm to create an SLOG file from a trace file

execption of R_L. Note, however, that elements added to R_L can instead be written directly to the output file. Since the number of elements added to the other lists is likely to be small, those lists can be maintained in memory. If for some reason the lists R_ℓ cannot be maintained in memory, separate temporary files may be used for them. In this case, some elements may need to be written to disk twice.

Reading all of the tree nodes for a given interval in an SLOG file consists of these steps:

1. Position at the end of the file.
2. Read the value of L.
3. Move backwards $2^L - 1$ (fixed-sized) records and read the directory.
4. For each node whose time interval intersects with the desired time interval, read the corresponding node.

The preceeding discussion has assumed that the trace file already contains the necessary records describing rectangles and other graphical objects. In practice, a trace file contains only events, along with enough information to generate the states, periods, and associations that we wish to display. We note that the process to convert events into displayable objects can be merged with the code to access the next record (exploiting the sequential access to the trace file needed by the algorithm in Figure 6), preserving the one-pass nature of this algorithm.

The above algorithm subdivides the data along the single dimension of time. Subdivisions in more dimensions are possible. For example, a 2-D (quad tree) decomposition that uses the vertical axis (process or thread) as the second dimension simplifies vertical scrolling and scalability in the number of separate threads. A 3-D (oct-tree) decomposition could use thread in process as the third coordinate (with process the second) or it could use record category as the third coordinate. The annotation model makes it easy to experiment with many different graphical representations of the same data.

4 Refinements

The SLOG API and algorithm can be enhanced in a number of ways.

4.1 Computing Data to Improve Display

During the pass over the original data that creates the SLOG annotations, additional data can be computed that can improve the quality and the performance (i.e., speed at which objects are drawn) of the display. Because the file of trace data may be large, it is important to perform these operations in a single pass.

1. Data for the preview can be collected during the execution of the SLOG algorithm. For example, the number of objects on each vertical line for each time interval (possibly using more intervals than leaves in the SLOG tree). Alternatives that weight by the area are possible. The point is that this data can easily be accumulated while reading the drawable objects. Combining objects by categories (e.g., communication or computation) for the preview may provide a more managable figure.

2. Since drawable objects may be nested (common with rectangles), having the "stack level" of an object is very helpful in the display. Since this is easily computed by the implementation of the SLOG input API, this information should be carried through to the display program. Of course, a stack level of "none" (not nested) or (perhaps?) unknown may be provided. An alternative value could be "depth" (commonly used by 2-d drawing programs to decide what object is drawn on top of other, overlapping (but not necessarily nested) objects). (Nesting level is more precisely what the SLOG Input API is computing. "Depth" is more likely to be something the Display Program might want to manipulate for itself. - RL)

3. In some cases, drawable objects, even states, will overlap without nesting. In this case, the objects can be laid out by computing a vertical offset that separates the objects in the y coordinate. This is particularly appropriate for rectangles (states).

   Question: what to do about arrows in this case? How are they attached to the "correct" state?

4. For very complex displays, a level-of-detail (LOD) feature may be important. For example, in cases where a large number of states crowd into a small interval of time could be represented by a single stippled rectangle rather than drawing a large number of (white-bordered in upshot/nupshot/jumpshot) rectangles. Zooming into this would eventually replace the stippled rectangle with the individual states. Note that LOD information is another type of annotation and can be computed at the same time as the other annotations by keeping track of adjacent states per vertical coordinate.

   A simple LOD algorithm would compare the location/extent of each drawable object to a virtual bitmap and render its bounding box on that bitmap. If the bounding box is less then 3 pixels in extent and there is already an object in that location on the virtual bitmap, then the object would be marked as undisplayable at that resolution, and the virtual bitmap would be marked as having a small object at the appropriate location.

   Objects are then sorted by LOD before being written out; that is, objects that are undisplayable at a particular level of detail are written after objects that are displayable at each level of detail. This allows a visualization program to read only the displayable objects, along with a way to represent "multiple objects here" (through the virtual bitmap).

   Question: Is there anyway to exploit this in the annotation form? That is, since we can't re-order the data, is there any way to exploit it?

   Note: Anthony has proposed an alternative form of LOD that permits continuous zooming of the image by adding "shadow object" information to each level in the tree, not just to the leaf nodes. One complication is that, since the shadow objects are more closely related to the drawable, there may need to be one set of shadow objects for each coordinate map (i.e., for each assignment of objects to timelines).

5. We can accumulate records until we reach a limit (based on memory size). We then end that leaf and begin a new leaf. Note that the duration of a leaf in this model is not constant; we might want to try

to encourage leaves to have the same duration. Note that in this model, we *do* get a (nearly) balanced tree because we simply group leave by twos, then group those nodes by twos, and so on. If instead leaves do have a duration that is of the form $T/2^k$ for some k, then different leaves may represent different values of k. Which should we do?

4.2 Enhanced Data Representations

1. Categories of categories can be added. This allows the display program to quickly select "all OS records" or "all point-to-point communication records." By defining a category as containing either a single drawable object or a list of categories, we can add this without changing the API used for the SLOG algorithm. That is each drawable object needs only specify the category that describes how it is rendered. (should we use a different term for categories of categories to emphasize that a drawable object belongs to a single category? Different term simplifies API. –RL) (WDG - I suggest category collection or category group, defined recursively as either a category or a category group.)

2. Category descriptions should include transparency as part of the color value.

3. The "rectangles" for states are really a "line with height" where the height may be assigned by the drawing program (possibly using the stack level information). Question: What about arrows, where the width of the arrow (or color?) is chosen to reflect the communication volume or other metric? Does this apply to single categories as well? For example, turn off all but one category, then assign a color to those states based on some value associated with each instance (We can get a value by calling a routine provided by the trace-file creator with the data-string).

4. Should there be a way to establish a relationship between instances of drawable objects? For example, the objects representing a particular send, receive, and the arrow between them (e.g., given a receive, find and blink the matching send). (The notion of "association" has proven thorny. Even if systems are truely likely to collect such data, it is not clear that it has to be more than just optional data. - RL)

5. We should consider separating the rendering from the sorting, allowing completely arbitrary rendering of displayed objects. This follows the approach of XML of separating all content description from presentation information. We do have to blur this a little because we use geometric information to sort the drawable objects, but properties like the color should be considered separate. In fact, the rendering method could be dynamically loaded by the viewer.

6. We've discussed the display of drawable objects. But these are usually organized by process, thread, or some other "container." How should the container be labeled and rendered? For example, should the "main thread" be rendered differently from other threads? User-generated versus system-generated threads?

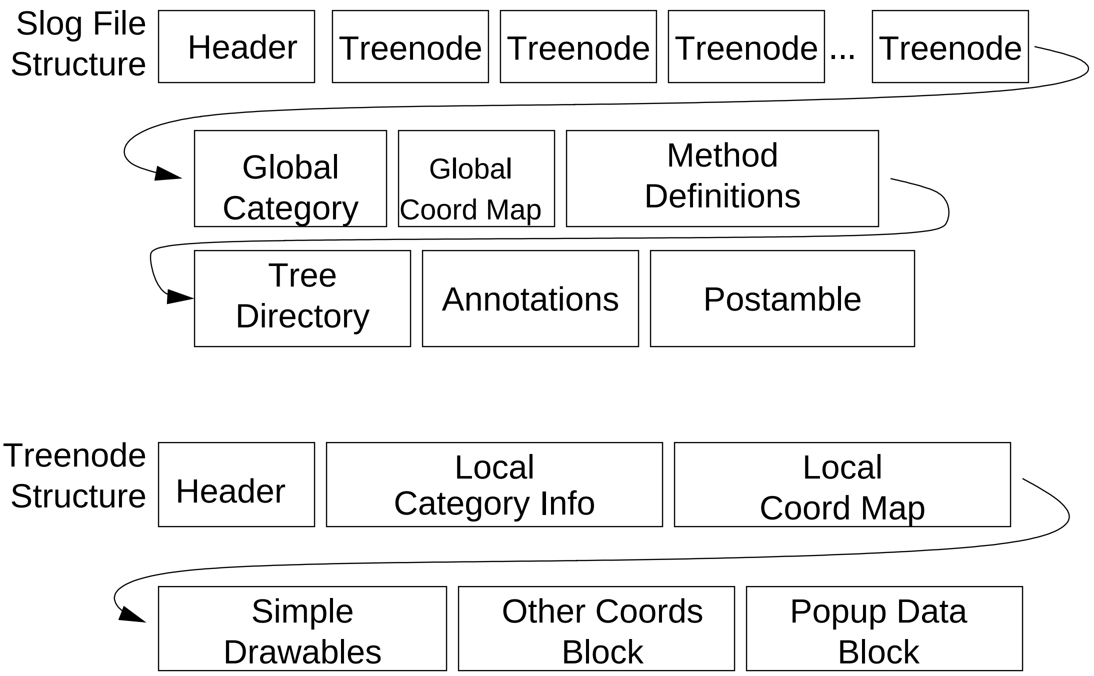

Figure 7: Block diagram of a complete SLOG2 file. Note that SLOG2 does not define a file format; this block diagram shows one possible organization to illustrate both the contents of the file and a structure that allows the creation of an SLOG2 file in a single pass.

4.3 Real-time File Generation

1. If the length of time T is not known, the algorithm in Figure 6 can still be used with a few changes. Specifically, the total number of levels is not known; as a "leaf" node list R_L fills up (reaches a maximum memory limit), a new time interval is created, possibly incrementing the number of levels. In other words, one starts with a single level ($L = 0$) and add levels as needed. The resulting tree will not necessarily be full. This is the reason for placing the directory at the end of the file, since in this case, the number of levels is not known a priori.

4.4 File Representation

Even though we have defined SLOG2 in terms of the APIs used to read and write the files, it is useful to discuss one possible representation of an SLOG2 file, such as that in Figure 7. Other representations are possible; the API describes the objects, not the way data is stored in a file.

The file format described here exploits the post mortum nature of the SLOG files. This allows us to collect data into logical groups, rather than forming it as a stream of records.

header The file header, containing information on the version of SLOG, name of the program and the user, and other data about the file.

treenode Block of data corresponding to R^j_ℓ.

global category info Information on the category definitions used in one or more treenode.

global coord map Information on the y coordinate mapping for coordinates in one or more treenode.

method definition Information on the methods that should be invoked for the popup data.

tree directory Block of offsets to the beginning of each treenode, along with the start and end times of each tree node. The offsets are 8-byte integers, in bytes, relative to the beginning of the file.

annotations Block of text that records the viewing history and user-provided annotations about the SLOG2 file. This block is at the end of the file (except for the postamble) so that annotations can be added to large files without rewriting the file.

postamble Contains five 4 byte integers indicating the location (relative to the end of the file) of the global-category-info, global-coord-map, method-definition, treedirectory and the annotation blocks. Four-byte integers are sufficient here because these objects are relatively small and are near the postamble. However, we could use 8-byte integers for consistency with treedirectory offsets.

The ordering of the blocks is chosen to make it easy to write an SLOG file with a single pass. With the exception of the treenode, each of these blocks is relatively small. It may be appropriate to use XML to describe each of these blocks, allowing XML-parsers and XML tools to look at the structure of SLOG2 files. An alternate form that stores the treenode data in a separate (binary) file could use XML for all of the other elements; this version would not require the postamble block.

The treenode is a special case. Because a treenode represents the smallest sensible unit of data, it makes sense to divide the data within a treenode into groups:

header This contains information on the location of the other parts of this treenode (e.g., the offsets of the starts of each block and sizes in terms of the number of records) along with information on the compression method (see below).

local category info Category definitions for drawable objects in this treenode only. In many cases, this block will be empty.

local coord map Coordinate mapping information for this treenode only. In many cases, this block will be empty. One example where it will not be empty is when individual threads are being created and destroyed, and some threads exist only for the duration of this treenode.

simple drawables The most basic information on the drawables. This is an array of fixed-length structures containing only the start and end times, category id, y coordinate, and offsets into the next two blocks. This allows rapid drawing of drawables that are simple states (rectangles).

other coords block This block contains all of the other coordinate values, both time and y, that are needed to describe more complex drawables.

popup data block This block contains all of the data that is used by the popup methods.

The choice of a small number of blocks each containing all entries of the various data elements simplifies the task of reading the data into the data structures within a display program. That is, a program can read the header (fixed length); allocate storage for the records based on information in the header, and read all of the data in a single operation.

Trace file formats, particularly for large amounts of data, often choose to define each data field with as few bits as possible in order to reduce file size. Because the SLOG file is (often) generated after the run of a program, we can use a different approach based on applying data compression to each treenode as a block. The treenode header indicates which type of compression has been used on the rest of the treenode. Among the possibilities are no compression, predefined static compression (in other words, the conventional trace-file approach based on defining the number of bits for each field), and dynamic compression, using for example the algorithms used in the `gzip` program (see, for example, [2, Section 9.1.2] for a description of the gzip algorithm). Dynamic compression allows us to eliminate the compromises of field lengths that static compression schemes must make. An additional advantage of dynamic compression, particularly when the file is accessed over a slow network, is that it can reduce the time to read the data, even when the time to decompress the file is included.

An SLOG Annotation file is almost the same. The only differences are (a) the block of treenodes contains only non-leaves of the tree, (b) the treedirectory describes the location of the treenodes that are leaves of the tree in terms of a location in another file, and (c) the treedirectory specifies the method used to access the treenodes. Note that if we use an XML representation of the SLOG file that places *all* of the treenodes into a separate file, the same format can be used for both complete SLOG files and SLOG annotation files.

4.5 Quantifying Data Motion

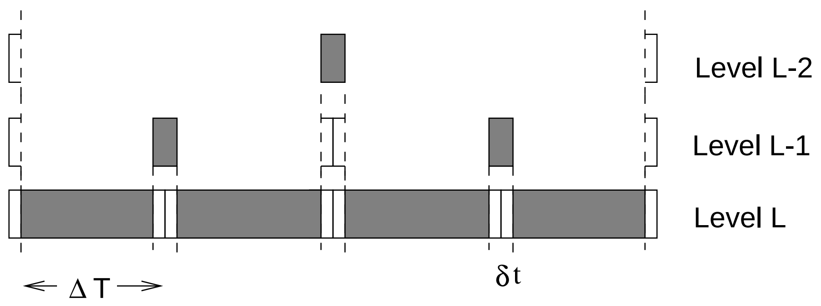

Figure 8: Any record whose time-center falls within the grey area is assigned to the corresponding node. Note that most of the records are assigned to leaf nodes.

One feature of the simple approach described in Section 3 is that some objects, even though they have very short duration, will be forced into lower (closer to the root) levels of the tree because they have the bad luck to cross the joint between two leaf boxes. For example, any drawable object that starts before $T/2$ and ends after $T/2$ will be forced into the root node, no matter how short its duration. In principle, this could create problems for the SLOG2 format by moving too many drawable objects out of the leaf nodes of the tree. Fortunately, we can show that in many cases this is not a problem; further, a small change to the format allows SLOG2 to handle all but very pathalogical cases.

Note that if there is a limit on the number of objects (boxes, arrows, etc.) that can cross any point in time, then the number of objects in the lower-level (non-leaf) bounding boxes can be bounded. If only states or periods are included, this can be often be effectively bounded. However, if drawable objects include connections between send and receive events or non-blocking I/O operations, the number of objects that can cross a particular time line can be very large.

We can estimate the number of records in each node for some simple situations. Consider the case where all records have the same duration δt and are uniformly distributed throughout the trace file. Consider first the algorithm in Figure 6, where the time intervals for each node on a level are disjoint (non-overlapping). Let the levels be $0, \ldots, L$, so that there are 2^L leaves. For each leaf interval of width ΔT_{min}, any record whose center starts $\frac{1}{2}\delta t$ past the beginning of the leaf's time interval and before $\frac{1}{2}\delta t$ before the end of the leaf's time interval will be placed within that interval. Thus, if there are N records, all but $N\delta t/\Delta T_{min}$ will be placed in the leaf nodes. Of the remaining records, $N\delta t/2\Delta T_{min}$ will be placed in the nodes at the next level, $N\delta t/4\Delta T_{min}$ in the next, down to level 0. This is illustrated in Figure 8.

The number of records read to display any time can be calculated as follows. Since the display of any time requires reading all time intervals that intersect that time, one interval on each level is read. The amount of data (not counting the leaf node) is simply

$$\sum_{k=0}^{L-1} \frac{N\delta t}{2^L \Delta T_{min}} = L\frac{N\delta t}{2^L \Delta T_{min}} = L\frac{N}{T}\delta t.$$

However, many of these records are unneeded. Consider the time $t = 0$. Only the first leaf must be read to provide all of the necessary data; these additional records are required only because they crossed the artificial boundaries that were defined between leaves. In the worst case, only

$$2\frac{N\delta t}{2^L \Delta T_{min}} = 2\frac{N}{T}\delta t.$$

records must be read; these correspond to the intervals at the left and right end of a leaf node. Thus, the number of unnecessary records that must be read is

$$(L-2)\frac{N}{T}\delta t.$$

In many cases, this will be a small number. For example, consider the case where the drawable objects do not overlap. Then $N\delta t \leq T$, and this expression is bounded by $L - 2$. Since $L = \log_2(T/\Delta T_{min})$, this number will rarely be large. If the degree of overlap is p, for example, there are p processes or threads, the number scales linearly with the degree of overlap. Further, the number of drawable objects at any time must be smaller that the number of available pixels (in the vertical dimension) for the graphical representation to be meaningful. This also provides a bound on the number of overlapping objects in a *useful* SLOG2 file.

However, if too many drawable objects of short duration cross the boundaries between tree nodes, the amount of unnecessary data that must be read could become large. To handle this case, we can generalize the bounding box notion to include overlapping bounding boxes. For example, if the bounding boxes at the leaves of the tree overlap by δt, then any drawable object of duration no more than δt can be placed in some leaf node. More complex distributions of duration can be analyzed and used to guide the amount of overlap at each level. This handles the pathalogical case mentioned above.

5 Experiments

Show some results, with file size and access times. Emphasize scalability.

A APIs

All of these APIs have some sense of “current” record. The simplicity of this outweighs possible problems with multi-threaded applications, particular since it is unlikely that application would use multiple threads to access or modify any of the trace files. Applications that need to use threads can use conventional thread synchronization techniques to guarantee correct behavior.

All routines are listed alphabetically.

B Input API for Reading from Foreign Trace Files

The input API for reading trace files assumes only that the file describes drawable objects; these may have to be assembled by the implementation of this API. Category and method information may or may not be present in the trace file (e.g., category information is present in clog files but method information is not). There is no coordinate map information here because no trace file currently provides that information.

TRACE_Category_head_t — Structure defining the basic information about a category

Synopsis

```
typedef struct {
  int category_index;
  int shape;
  int red, green, blue, alpha;
```

```
} TRACE_Category_head_t;
```

category_index
integer value by which records will identify themselves as belonging to this category

shape
Shape of the category. This is an integer defined by the drawing program; the value 0 is reserved for a basic state (a rectangle along a timeline)

red, green, blue
Color of the shape; each is in the range [0,255]

alpha
Transparency value, in the range of [0,255]. Some display programs may ignore this value.

TRACE_Close — Close a trace file

Synopsis

```
int TRACE_Close( TRACE_file *fp )
```

Input Parameter

fp Pointer to a trace file handle

Notes

The pointer `fp` is set to null on a successful close.

TRACE_Get_next_category — Get the next category description

Synopsis

```
int TRACE_Get_next_category( const TRACE_file fp,
                             TRACE_Category_head_t *head,
         char *label_base, int *label_pos, const int max_label,
         char *legend_base, int *legend_pos,
         const int max_legend,
         char category_methods[][],
```

```
char category_method_extra[][] )
```

Input Parameter

fp Trace file handle

Input/Output Parameters

label_pos On input, the first available position in `label_base` On output, changed to indicate the new first available position.

legend_pos On input, the first available position in `legend_base` On output, changed to indicate the new first available position.

Output Parameters

head Contains basic category info (see the description of `TRACE_Category_head_t`)

label_base Pointer to storage to hold label information

max_label Size of `label_base`

legend_base Pointer to storage to hold legend information

max_legend Size of `legend_base`

category_methods
Null-terminated array of null-terminated strings describing methods used to process record-specific data.

category_method_extra
Extra data for each method

Notes

The interface to this (and similar routines such as `TRACE_Get_next_drawable`) is designed to give flexibility in how data is read. See `SLOG2_Get_next_category` for more details.

TRACE_Get_next_drawable — Get the next drawable

Synopsis

```
int TRACE_Get_next_drawable( const TRACE_file fp,
```

```
double *starttime, double *endtime,
int *category_index,
int *nt_coords, double *tcoord_base,
int *tcoord_pos, const int tcoord_max,
int *ny_coords, int *ycoord_base,
int *ycoord_pos, const int ycoord_max,
char *byte_base, int *byte_pos,
const int byte_max )
```

Input Parameter

fp Trace file handle
tcoord_max Size of `tcoord_base`
ycoord_max Size of `ycoord_base`
byte_max Size of `byte_base`

Input/Output Parameters

tcoord_pos On input, the first free location in `tcoord_base`. Updated on output to the new first free location.
ycoord_pos The same, for `ycoord_base`
byte_pos The same, for `byte_base`

Output Parameters

starttime, endtime
time range for drawable
category_index
index of the category that this drawable belongs to
nt_coords Number of time coordinates
tcoord_base pointer to storage to hold time coordinates
ycoord_base pointer to storage to hold y coordinates
byte_base pointer to storage to hold bytes

Question

Should the information that is also returned by `TRACE_Peek_next_drawable` also be returned by this routine?

TRACE_Get_next_method — Get the next method description

Synopsis

```
int TRACE_Get_next_method( const TRACE_file fp,
                           char method_name[], char method_extra[],
                           int *method_id )
```

Input Parameter

fp Trace file handle

Output Parameters

method_name
method_extra
method_id

Note

A typical trace file may have *no* methods.

Question

Should methods have ids so that categories can refer to them by id instead of by name?
How do we ensure that the data areas are large enough?

TRACE_Get_position — Return the current position in an trace file

Synopsis

```
int64_t TRACE_Get_position( TRACE_file fp )
```

Input Parameter

fp Trace 2 file handle

Return Value

Current position of file. Returns `-1` on error.

Notes

This routine and `TRACE_Set_position` are used in the construction of an annotated Slog file. In an annotated Slog file, the Slog file records the location in the original trace file of the records, rather than making a copy of the records.

TRACE_Get_total_time — Return the time range covered by an trace file

Synopsis

```
int TRACE_Get_total_time( const TRACE_file fp,
                          double *starttime, double *endtime )
```

Input Parameter

fp Slog 2 file handle

Output Parameters

starttime Time when log file begins (no event before this time)
endtime Time when log file end (no event after this time)

Questions

Do we want to require this? In some cases, it may be difficult to return this time.

TRACE_Open — Open a trace file for input

Synopsis

```
TRACE_file TRACE_Open( const char filename[] )
```

Input Parameter

filename Name of file to open.

Return Value

Trace file handle

Notes

The file is opened for reading only.

Questions

This does not handle the case where the trace file is actually a collection of files or a non-file (e.g., a socket). An easy fix for this is to change `filename` to `filespec` and define that as an arbitrary string. Then any string tha the `TRACE_Open` command accepts can be used. Since the Slog program will only pass this string through (e.g., from the command-line to this call), it need not be a file name.
Possible interpretations of `filespec` include a filename, an indirect file (i.e., a file that contains the names of other files), a colon separated list of files (i.e., `file1:file2:file3`), a file pattern (i.e., `file%d`), any of the above along with other options (for the trace file reader), such as limits on the time range or node numbers to accept, or even a shell command (i.e., `find . -name '*.log'`).

TRACE_Peek_next_drawable — Peek at the next drawable

Synopsis

```
int TRACE_Peek_next_drawable( const TRACE_file fp,
                              double *starttime, double *endtime,
                              int *nt_coords, int *ny_coords, int *n_bytes )
```

Input Parameter

fp Trace file handle

Output Parameters

starttime, endtime
time range for drawable
nt_coords Number of time coordinates
ny_coords Number of y coordinates
n_bytes Number of data bytes

Notes

This routine allows the calling code to allocate space for the variable-length data in a drawable before calling `TRACE_Get_next_drawable`. The time range allows the SLOG2 algorithm to determine which treenode a drawable should be placed in (which may influence where in memory the data is read by `TRACE_Get_next_drawable`).

TRACE_Peek_next_kind — Determine the kind of the next record

Synopsis

```
TRACE_Rec_Kind_t TRACE_Peek_next_kind( const TRACE_file fp )
```

Input Parameter

fp Trace file handle

Return Value

Type of next record. The kind `TRACE_EOF`, which has the value `0`, is returned at end-of-file.

Notes

The structure and ordering of data in a foreign trace file is not defined. This routine allows us to find out the type of the next record and then use the appropriate `TRACE_Get_xxx` routine to read it. A high-performance implementation of these routines will likely use buffered I/O.

TRACE_Rec_Kind_t — Types of records in a trace file

Synopsis

```
typedef enum {TRACE_EOF=0,
              TRACE_DRAWABLE,
              TRACE_CATEGORY,
              TRACE_METHOD} TRACE_Rec_Kind_t;
```

TRACE_Set_position — Set the current position of a trace file

Synopsis

```
void TRACE_Set_position( TRACE_file fp, int64_t offset )
```

Input Parameters

fp	Trace file handle
offset	Position to set file at

C SLOG2 Output API

The SLOG2 Output API is designed to be used by the Slog program to create Slog 2 files. This API is organized around the concepts defined in Section 3, particularly the tree nodes and tree directory.

The SLOG2 Output API does *not* define the order of data in an SLOG2 file. However, the following structure (see also Figure 7) allows for both single-pass creation and relatively easy read access to the resulting file.

header

- **cookie** type of file, e.g., SLOG annotation file, SLOG complete file
- **version** to ensure compatibility with Display program
- **textlen** length of following description field
- **description** text string describing the file (e.g., "trace of a.out on July 3, 2000, on big blue with dataset 3")
- **filespec** file specification string used with `TRACE_Open`. This field is required for the Slog annotation format and optional otherwise.

tree nodes the slog records. This is the bulk of the data in the file

category information set of category description records

coordinate maps set of coordinate maps; for example, for process, thread, and MPI rank views.

method definitions information about and for the methods used to interpret and/or display the drawable objects.

tree directory The hierarchy of describing the tree of nodes

annotations Any user-defined annotations added to an SLOG2 file

postamble

- **category_offset** offset from end to category information
- **tree_offset** offset from end back to beginning of tree
- **annote_offset** offset from the end to the annotations
- **coord_offset** offset from the end to the coordinate maps
- **method_offset** offset from the end to the method descriptions

This file can be created in one pass, since the tree information is added at the end of the file, and the offsets in the postamble allow a program to seek to the tree and display data without reading through the file. The tree nodes have file offsets in them (see the description of `SLOG_Treenode`) so then one can use them to seek to specific frames in the file.

A category record includes

```
category id (small int)
category name length
category name (string)
shape (rectangle, polygon, arrow, set-of-rectangles, etc. )
         set-of rects is for "bebit handling"
color (r,g,b,alpha?)   default, can be changed by display
textlength of following text
text for display (sprintf format, with %s's, \n's)
methods to use (methods are the popup methods (e.g., popup args,
popup source code, etc.)
```

A data record has

```
category
...  (as in slog-read api)

the instance-specific data is the concatenation of
null-terminated strings, to be merged with the category-specific
sprintf string.  Note that regardless of the what is stored in
the file, what is presented back through the API is string data.
```

The coordinate class definition allows an SLOG file to define several mappings of y coordinate (the non-time coordinate) to an integer index (the timeline index), along with a timeline label. This allows unified process/thread/node ids to be used in the SLOG2 file, and for a single SLOG2 file to provide mappings to node, process, or thread views, with an arbitrary labeling. If no coordinate class defintion is provided, the identity mapping is used (coordinates used as defined in the file, with the timeline labels defined by the character version of the value).

The method definition passes to the invocation of each method the following three values:

1. The `method_extra` string (every use of that method)
2. The category-specified `extra_data` string (every use of that method by drawable objects in that category)
3. The byte string on the drawable object.

For example, a method that displays the arguments for a method might use a category-specified `extra_data` string containing, for example, `"count = %d\nrank = %d\ntag = %d"`. A method that pops up a source code box might have a `method_extra` string that contains the name of the executable file; the category-specified string would be empty, and the byte string of the drawable object would contain the address in the program where the drawable object was logged.

Methods are invoked in the order they are specified to the category. They return the number of bytes that they have read from the byte string. The byte string passed to each method always starts at the next unused byte.

The definition field in a method specification may be used to specify the location of a Java class or DLL that implements the method, or a URL of a page that describes the behavior of the method.

SLOG2_Write_category — Add a category description to an Slog file

Synopsis

```
int SLOG2_Write_category( const SLOG2_File fp,
                          const SLOG2_Category_head *head,
                          int category_text_length,
                          const char category_text[],
                          const char methods[][],
                          const char method_extra[][] )
```

Input Parameters

fp Slog file handle + head - Contains basic category info (see the description of `SLOG2_Category_head_t`)

category_text_length Number of characters in `category_text`

category_text Text describing the category, suitable for use in a legend

methods Null-terminated array of null-terminated strings describing methods used to process record-specific data.

method_extra Extra data for each method.

Notes

A `category_index` of `-1` may be used to indicate that no more category descriptions will be made. Once that has been done, it is invalid to call `SLOG2_Write_category` on that file handle.
Methods provide a way for a category to specify how to process data that each record provides. For example, with each record generated by a call to a routine, there could be data consisting of the program counter, following by the arguments to the routine. The category could specify two methods for this; the first would read the program counter value and pop up a source-code browser; the second method could pop up a box containing text listing the arguments to the routine (by name). More complex methods, for example, could pop up graphical displays of array arguments to the routine.

Module

SLOG2 Output

SLOG2_Write_coord_class — Write out a coordinate class description

Synopsis

```
int SLOG2_Write_coord_class( const SLOG2_File fp, const char class_name[],
                             int ncoords,
                             const int in_coord_val[], int out_coord_val[],
                             const char coord_label[][] )
```

Input Parameters

fp Slog file handle

class_name Name of the coordinate class

ncoords Number of coordinates in `in_coord_val` and `out_coord_val`

in_coord_val, out_coord_val
Coordinate pairs to match. For `j` between `0` and `ncoords-1`, the value of `in_coord_val[j]`, specified as a `y` coordinate in a record, is mapped to the value `out_coord_val[j]`.

coord_label Text describing the coordinate, suitable for use in a legend.

Notes

This routine allows an slog file to define several different views of the data. For example, a record may contain, for the `y` coordinate, the node, process, and thread id that generated the record. Coordinate classes allow an slog file to define different ways to map the node/process/thread id into a `y` coordinate. Using three calls to this routine, an Slog file could define a node, process, and thread view of the data. The `coord_label` allows a display program, such as Jumpshot, to show the user what different coordinate views are available.
If no coordinate class is defined, a default coordinate class is defined that provides the identity mapping (the output coordinate values are the same as the input values; e.g., the `y` coordinates in the drawable objects are used directly).
Note that the code that makes use of the coordinate map needs to provide an efficient way to locate the `in_coord_val` so that the correct `out_coord_val` can be found. For example, a hash function may

be used. As a special case, there may be other predefined maps besides the identity map (e.g., a simple thread_id to out_coord_val mapping).

Rationale

One reason for this routine is that the thread-view is difficult for a display program to manage, particularly if threads are created and destroyed frequently. The program in the best position to create a mapping of thread ids to `y` coordinates is the program that creates the Slog2 file, since it must read every record.

Question

This assumes that we can create a single int containing the original `y` coordinate. For a node/process/thread value, we might have a more than an int's worth of bits. The reason that we leave it at an int it that there will never be more than an int's worth of *significant* bits; that is, there will never be more than 4 billion distinct `y` values.

Module

SLOG2 Output

SLOG2_Write_dir — Write an entire Slog2 tree directory

Synopsis

```
int SLOG2_Write_dir( const SLOG2_File fp,
                    const SLOG2_Treedir_node dir[],
                    int n_nodes )
```

Input Parameters

fp	slog2 file handle
dir	Tree directory
n_nodes	Number of nodes in `dir`

Module

SLOG2 Output

SLOG2_Write_drawable_object — Add a drawable object to an slog file

Synopsis

```
int SLOG2_Write_drawable_object( const SLOG2_File fp,
                       double starttime, double endtime,
                       int category_index,
                       int nt_coords, const double t_coord[],
                       int ny_coords, const int y_coord[],
                       int ntext, const char text[] )
```

Input Parameters

fp
Slog file handle

starttime, endtime
time range for object

category_index
Category that this object belongs to

nt_coords
Number of time (`x`) coordinates. May be `0` if `starttime` and `endtime` are sufficient

t_coords
Time coordinates

ny_coords
Number of y coordinates. Must be at least `1`.

y_coord
Y coordinates

ntext
Number of bytes of data for this object. May be `0`

text
Data bytes for this object. This data is input to the methods defined by the category.

Notes

The `y` values may be simple values between `0` and a small integer or they may be more complicated integers, representing, for example, a node/process/thread id tuple. In the latter case, one or more coordinate maps should be defined with `SLOG2_Write_coord_class`.

Module

SLOG2 Output

SLOG2_Write_end_treenode — End writing a tree node

Synopsis

```
int SLOG2_Write_end_treenode( const SLOG2_File fp )
```

Input Parameter

fp	Slog file handle

Module

SLOG2 Output

Question

Should this return a position or a length for the directory?

SLOG2_Write_header — Write the Slog2 file header

Synopsis

```
int SLOG2_Write_header( const SLOG2_File fp,
                        const char description[] )
```

Input Parameters

fp	Slog file handle
description	Null terminated description text

Notes

This routine should be called before any other slog2 output routines. This version also writes the cookie and version number of the library to the file header.

Questions

If we require that this be called first, should we merge it with Open?

Module

SLOG2 Output

SLOG2_Write_method_desc — Add a method description to an Slog file

Synopsis

```
int SLOG2_Write_method_desc( const SLOG2_File fp,
                             int ntext, const char text[],
                             const char common_extra[],
                             const char definition[] )
```

Input Parameters

fp Slog file handle
ntext Number of characters in `text`
text name of the method
common_extra
Common data used by this method
definition Definition of the method. This may, for example, be the URL for the Java class that implements the method.

Notes

This is intended to provide a common method description for methods that are used by many category descriptions. E.g., the source-code browser pop up method needs the name and location of the executable and the location of the source files, along with the name of the program to find the correct point in the source file. Rather than have *every* category description include this information, this routine allows a named method to share data across all categories.

Module

SLOG2 Output

Question

Do we need `ntext`, or should we assume null-terminated strings?

SLOG2_Write_start_treenode — Begin writing a tree node

Synopsis

```
int SLOG2_Write_start_treenode( const SLOG2_File fp )
```

Input Parameter

fp	Slog file handle

Notes

This routine is called before writing the contents of a single tree node. When the treenode is complete, `SLOG2_Write_end_treenode` must be called.

Module

SLOG2 Output

Question

Should this return a position (`int64_t`) for the directory?
Currently, there are no routines for writing the local category or coordinate map information (local as in local to the treenode). Do we need them?

D SLOG2 Input API

The SLOG2 Input API is used to read an Slog file, and is intended for use by programs such as Jumpshot. This list is incomplete, and does not match all the capabilities of the slog2 output api.

SLOG2_Close — Close an Slog 2 file

Synopsis

```
int SLOG2_Close( SLOG2_File *fp )
```

Input Parameter

fp	Pointer to an Slog 2 file handle.

Notes

`fp` is set to null by this routine.

SLOG2_Get_drawable_text — Get a text string describing a particular drawable object

Synopsis

```
void SLOG2_Get_drawable_text( SLOG2_File fp,
                              int category, const char *drawable_text,
                              char *instance_text, const int text_max )
```

Input Parameters

fp	Slog 2 file handle
category	Category index of drawable
drawable_text	Text from drawable
text_max	Maximum size of `instance_text`

Output Parameter

instance_text Text describing this particular drawable, as formatted acording to the rules of the category.

Question

Is this superceeded by the general category method approach, or is this an important special case? E.g., is this basically the default method case?

SLOG2_Get_header — Read the header of an Slog 2 file

Synopsis

```
int SLOG2_Get_header( const SLOG2_File fp,
                      char description[], int maxdesclen,
                      int *version )
```

Input Parameters

fp Slog 2 file handle
maxdesclen Length of the array `description`

Output Parameters

description Character string describing the SLOG file
version Version number of SLOG2 file

Return Value

Zero on success. If `maxdesclen` is too small to hold the description, the return value is the negative of the length needed. A positive value is a standard Unix `errno` value corresponding to the error.

Question

Do we want a routine that returns values of `maxdesclen` (and the size of the directory tree etc.)? What error value do we return if the version number in the file doesn't match the one that the library is expecting?

SLOG2_Get_next_category — Read the next category description

Synopsis

```
int SLOG2_Get_next_category( const SLOG2_File fp,
                             SLOG_Category_head_t *head,
           char *text_base, int *text_pos, const int max_text,
           char *legend_base, int *legend_pos,
           const int max_legend )
```

category_index = -1 means no more categories

Notes

This routine is structured this way to give the maximum flexibility in reading the data. For example, one approach could define a category structure, such as

```
typedef struct {
    SLOG2_Category_head_t head;
    char text[MAX_TEXT];
    char legend[MAX_LEGEND];
 } Category_t;
```

and call this routine as

```
Category_t *p;
int pos = 0, lpos = 0;
p = (Category_t *)malloc( sizeof(Category_t) );
SLOG2_Get_next_category( fp, &p->head
                       p->text, &pos, MAX_TEXT,
                         p->legend, &lpos, MAX_LEGEND );
```

Alternately, the categories can be stored in common storage, particularly for the text and legend. For example,

```
typedef struct {
    SLOG2_Category_head_t head;
    int text_pos, text_len, legend_pos, legend_len }
    Category_t;

  Category_t p[MAX_CATEGORY];
  char       text[MAX_TOTAL_TEXT], legend[MAX_LEGEND_TEXT];
  int pos = 0, lpos = 0;
  int catnum;
  ...
  p[catnum].text_pos = pos;
  p[catnum].legend_pos = lpos;
  SLOG2_Get_next_category( fp, &p[catnum].head,
                          text, &pos, MAX_TOTAL_TEXT,
                            legend, &lpos, MAX_LEGEND_TEXT );
  p[catnum].text_len = pos - p[catnum].text_pos;
  p[catnum].legend_len = lpos - p[catnum].legend_pos;
  catnum ++;
```

The reason for this flexibility is to allow fixed-length representations for most of a category and allow the code reading the slog 2 file to decide where to place the variable-length data.

Rationale

The `SLOG_Category_head_t` provides a reasonable comprimise between specifying each element of the head (e.g., six separate arguments for each of the fields in the `SLOG_Category_head_t`) and defining a single category structure that contains the text and legend fields.

SLOG2_Get_next_drawable — Get the next drawable object

Synopsis

```
int SLOG2_Get_next_drawable( const SLOG2_File fp,
                             double *starttime, double *endtime,
                             int *category_index,
                             int *ncoords, double *coord_base,
                             int *coord_pos,
                             const int coord_max,
                             char *text_base, int *text_pos,
```

```
const int text_max )
```

Input Parameters

fp Slog 2 file handle
coord_base Pointer to storage for coordinates (see `coord_pos`)
coord_max Size of `coord_base`
text_base Pointer to storage for text (see `text_base`)
text_max Size of `text_base`

In/Out Parameters

coord_pos
text_post

Output Parameters

starttime,endtime
Beginning and endtime time of drawable
category_index
Category that this drawable belongs to
ncoords Number of coordinates for this drawable

Notes

SLOG2_Get_position — Return the current position in an Slog2 file

Synopsis

```
int64_t SLOG2_Get_position( const SLOG2_File fp )
```

Input Parameter

fp Slog 2 file handle

Return Value

Current position of file.

Question

The slog2 output api also has this routine. Do they need to be different?

SLOG2_Get_total_time — Return the time range covered by an Slog 2 file

Synopsis

```
int SLOG2_Get_total_time( const SLOG2_File fp,
                          double *starttime, double *endtime )
```

Input Parameter

fp	Slog 2 file handle

Output Parameters

starttime	Time when log file begins (no event before this time)
endtime	Time when log file end (no event after this time)

SLOG2_Open — Open an Slog 2 file

Synopsis

```
SLOG2_File SLOG2_Open( const char filename[], char mode )
```

Input Parameters

filename	Name of the file to read or write
mode	`'r'` for reading and `'w'` for writing

Return Value

A handle to an SLOG2 file.

Question

The `mode` was added since the SLOG2 output API defines the same routine. Do we want to use the same routine (and make these all part of a single slog2 library) or do we want different routines? If so, we may want a different prefix, e.g., slog2in and slog2out instead of just slog2.

SLOG2_Peek_next_drawable — Peek at the next drawable

Synopsis

```
int SLOG2_Peek_next_drawable( const SLOG2_File fp,
                              double *starttime, double *endtime,
                              int *nt_coords, int *ny_coords,
                              int *n_text )
```

Input Parameter

fp	Slog 2 file handle

Output Parameters

starttime, endtime	time limits of drawable
nt_coords	number of t coordinates
ny_coords	number of y coordinates
n_text	number of bytes in text

SLOG2_Read_dir — Read an Slog 2 directory structure

Synopsis

```
int SLOG2_Read_dir( const SLOG2_File fp, SLOG2_Treenode dir[],
                   int maxtree, int *n_nodes )
```

Input Parameters

fp	Slog 2 file handle
maxtree	Size of the array `dir`

Output Parameters

dir	Array describing the Slog 2 directory (see Notes)
n_nodes	The number of nodes in `dir`

Notes

An Slog 2 file is described by a tree-structured directory. This call returns the directory in an array provided by the user. Each node in the tree is a struct of type `SLOG2_Treenode`. The root is `dir[0]`; the fields `lchild` and `rchild` give the indices in `dir` of the left and right children respectively. A parent of `-1` indicates the root.

SLOG2_Set_position — Set the current position of an Slog2 file

Synopsis

```
void SLOG2_Set_position( const SLOG2_File fp, int64_t offset )
```

Input Parameters

fp	Slog 2 file handle
offset	Position to set file at

E Jumpshot Plans

While Jumpshot is not part of Slog2, the Jumpshot program is the program that we are planning to use to display Slog files, and the Slog design includes features designed to make the implementation of Jumpshot easier.

Some of the features that the next Jumpshot will have include

1. Pluggable modules for most graphical objects:
   (a) Preview module
   (b) Display module; specifically, each category type can specify one or more methods that are used to display the records belonging to that category. We should also consider a canvas module that manages the display of the timelines.
   (c) Analysis modules. This will allow the user to provide a module that can analyze the records and display information or generate a different graphical interpretation. One example is a module that causes excessively long (slow) communications to blink.

   Of course, Jumpshot will provide default modules for all of these; the point of making them pluggable is both to allow user customization and easy extension and to enable research into other display approaches.
2. Smooth scrolling in time, even across tree nodes.
3. Today's version of Java/Swing (e.g., 1.3?). With any luck, Java and Swing will stop changing in backward-incompatible ways.
4. Runnable through a web browser.
5. Data structures designed to work efficiently with Java.

F SLOG1 to SLOG2 Transition Plans

In Slog1, there is an API to write an Slog file, directly from a source file such as UTE or clog. None of the APIs described above fits that model. This is a result of both the desire to create an annoted-only slog file (e.g., one that contains *only* the scalablity information, not a copy of the original data records) and the reorganization of the Slog file to be more block oriented.

An API to that provides data records to the Slog2 program (i.e., the program that writes the Slog2 file using the Slog2 output API) could also be written; it should probably be similar to the current API used to write Slog1 files from UTE files. However, we will need to modify that API so that the short-term comprimises used to deliver a working version are replaced with the more general Slog2 features. Specifically, the category and method-based processing of byte-strings associated with each drawable object will allow handling of arbitrary data, including 64-bit addresses and non-integer data (e.g., file names). We may also want to provide some coordinate map information.

The steps for developing the code for SLOG2 follow:

1. Write a version of the tracefile input API for the MPICH clog format.
2. Write a version of the SLOG2 Output API and implement the SLOG2 program as described in Section 3.
3. Write a corresponding verison of the SLOG2 Input API and several diagnostic programs (e.g., print the contents of the SLOG2 file).
4. Develop an all-new version of Jumpshot as described in Appendix E.
5. Develop annotated Slog2 varients of the SLOG2 APIs and make Jumpshot work with them.

Concurrently with these steps, a plan will be developed to interface UTE to SLOG2.

References

[1] TimeScan Multiprocess Event Analyzer. `http://www.etnus.com/products/timescan`.

[2] Darrel Hankerson, Greg A. Harris, and Jr. Peter D. Johnson. *Introduction to Information Theory and Data Compression*. CRC Press, 1998.

[3] Michael T. Heath and Jennifer Etheridge Finger. Paragraph: A performance visualization tool for MPI. `http://www.csar.uiuc.edu/software/paragraph`.

[4] IBM. *IBM Parallel Environment for AIX*, volume 2, chapter 3. IBM, 1998. http://www.s390.ibm.com/bookmgr-cgi/bookmgr.cmd/books/PEOP2240/CONTENTS.

[5] Ewing Lusk. Performance visualization for parallel programs. *Theoretica Chimica Acta*, 84:377–384, 1993.

[6] A. Malony, D. Hammerslag, and D. Jablonowski. TraceView: A trace visualization tool. *IEEE-Software*, 8(5):19–28, September 1991.

[7] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[8] Vampir 2.0 – Visualization and Analysis of MPI Programs. `http://www.pallas.de/pages/vampir.htm`.

[9] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In Anthony Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society, 1993.

[10] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.