Scientific Programming 16 (2008) 155–165
DOI 10.3233/SPR-2008-0252
IOS Press

An efficient format for nearly constant-time access to arbitrary time intervals in large trace files

Anthony Chan *, William Gropp ** and Ewing Lusk
Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

Abstract. A powerful method to aid in understanding the performance of parallel applications uses log or trace files containing time-stamped events and states (pairs of events). These trace files can be very large, often hundreds or even thousands of megabytes. Because of the cost of accessing and displaying such files, other methods are often used that reduce the size of the tracefiles at the cost of sacrificing detail or other information.

This paper describes a hierarchical trace file format that provides for display of an arbitrary time window in a time independent of the total size of the file and roughly proportional to the number of events within the time window. This format eliminates the need to sacrifice data to achieve a smaller trace file size (since storage is inexpensive, it is necessary only to make efficient use of bandwidth to that storage). The format can be used to organize a trace file or to create a separate file of *annotations* that may be used with conventional trace files. We present an analysis of the time to access all of the events relevant to an interval of time and we describe experiments demonstrating the performance of this file format.

Keywords: Trace file, message passing interface, performance visualization

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 execution of the program. A trace file contains several basic elements. Typically, these are generated during the execution of a program by short code sequences (so as to minimize the perturbation of the execution caused by the tracing [16]) and are written either to disk (buffered, of course) or to memory as they are generated. Trace files typically contain sequences of *events*; an event has a timestamp and some data. Pairs of events may be used to define a *state* or *duration*; these often represent the entry to and exit from a routine or a block of code. 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 [2–4,8–11,13–15,18–20,22] 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 collected into a trace file can become extremely large, exceeding hundreds or even thousands 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, sometimes one must 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. One approach that has proved effective is the GANNT chart, 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. Many tools (see [22], for example) augment this view with further detail, such as arrows to show messages

*Current address: ASCI Flash Center, University of Chicago, Chicago, IL, USA.
**Corresponding author (current address): William Gropp, Department of Computer Science, University of Illinois at Urbana-Champaign, IL, USA.

and popup windows to display detail data on process states or messages. Even when the simplest form of GANNT chart is being displayed, however, the basic problem of *scalability* arises, and the issues discussed in this paper can be thought of in this context.

Therefore, we state the general problem as follows. We assume that a parallel program produces as it runs a large volume of data on program behavior, including (possibly nested) states of processes varying over time. We make no assumptions about the maximum lengths (in time) of those states, though we do assume that most states are short in time. We wish to design a file format 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 point in time depend on the number of graphical objects to be displayed there and not on the total amount of program data nor on the particular time chosen. A rough approximation of its appearance (minus the colors) is shown in Fig. 1.

Section 2 describes the software context of our work, explains the problem in a little more detail, and identifies some related research. Section 3 describes the design of the SLOG2 file format, including a one-pass algorithm for creating the SLOG2 file from data presented in timestamp order. A one-pass algorithm is essential because we assume that the size of the trace files is extremely large. Section 4 shows the results of some experiments with our implementation of SLOG2. Section 5 analyzes the amount of data that must be read to correctly render an arbitrary interval of time, under reasonable assumptions about the distribution of data in the trace file. Section 6 describes some enhancements that further improve the SLOG2 file, along with an analysis of the design choices. Section 7 gives a summary of the paper.

2. Background

In this section we provide some context that motivated this work, describe the nub of the problem, and discuss related efforts in the area of scalable interpretation of trace files.

2.1. Motivation

Our motivation comes from our attempts over the years to improve the usability and scalability of our trace file visualization tool called, in its current incarnation, Jumpshot [22]. Jumpshot is the display component of a standard pipeline for trace file visualization as shown in Fig. 2.

In the tools that accompany the MPICH2 implementation of MPI, the elements of this pipeline were originally as follows:

logging: The MPE library provides functions that allow the efficient buffering in memory (with spill to disk if necessary) of timestamped events. These typically record the beginning and ending of program states. This library is accessible to both applications and an MPI profiling library for recording all MPI calls as states. Message events record sizes and tags for messages between processes or threads. Clock differences are corrected, and the event streams from various processes are merged into a single file.

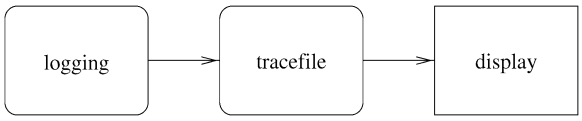

Fig. 2. Standard pipeline for trace file generation and display.

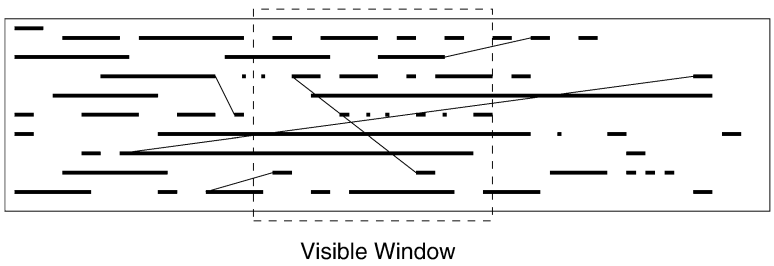

Fig. 1. Example of a trace file display. Only the data within the dashed box is displayed; the visualization program must render all data as if the entire data is displayed, clipped to the dashed box. This includes the rectangles that enter and exit the dashed box and the lines connecting the rectangles.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

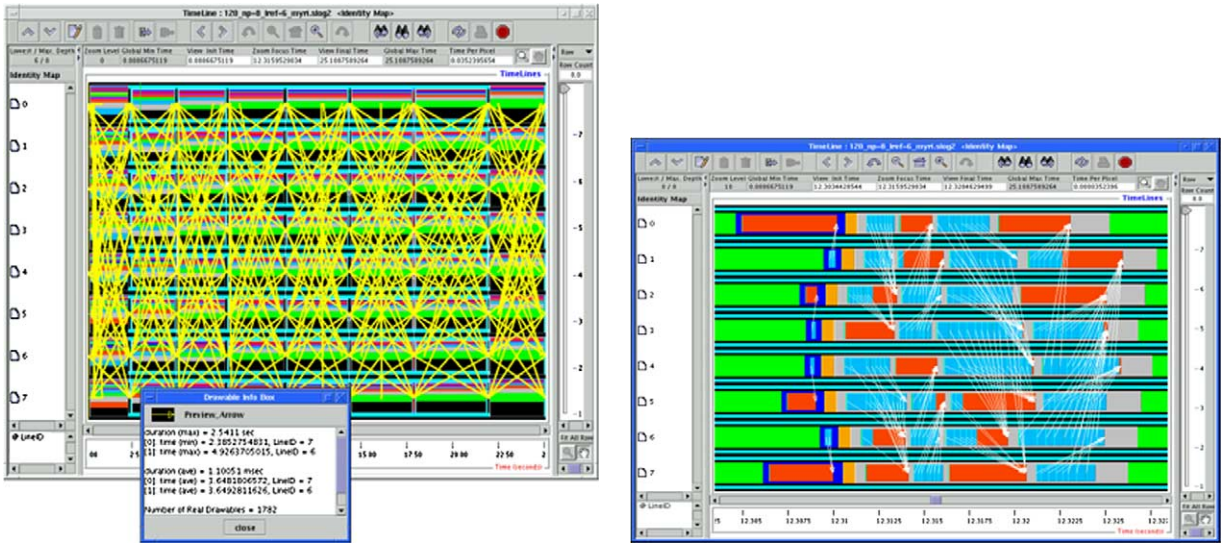

Fig. 3. On the left, Jumpshot summary view. Each line represents thousands of messages, and each block represents the states in that time interval, proportionally represented. On the right, zoomed-in view of the same trace file, by a factor of roughly 1000.

Fig. 4. Longer pipeline with conversion of trace file to SLOG file.

tracefile: The file written by MPE logging is in a format we call CLOG. It is a more or less standard file of timestamped events and message information.

display: Jumpshot is a full-featured trace visualization program. It can show a high-level, summary view (see Section 3.4) of the entire trace and zoom in by factors of thousands to scroll through details of program behavior. Jumpshot screenshots are shown in Fig. 3. The one on the right is the Jumpshot version of Fig. 1.

Our desire to have Jumpshot provide constant, quick, interactive response has motivated the work reported here. An important step was to realize that the same file structure could not be optimal for both the logging process and the display process, particularly for large trace files. Hence we introduced a separate file format (SLOG) to provide for scalable operation of a display program, leading to a system like that shown in Fig. 4.

The SLOG file contains drawable objects such as rectangles representing states and arrows representing messages as well as individual events, along with an index to allow direct access to internal places in the file. SLOG2 is the second generation of this approach. While the MPE logging mechanism and the Jumpshot display program are of some interest in their own right, this paper is about the SLOG2 file format. SLOG2 files are accessed by a CLOG-to-SLOG2 convertor and Jumpshot through a well-defined interface, so the format can be used in other contexts.

2.2. *The tricky part of the problem*

What makes zooming into files (either events or states) challenging is that the obvious division of the file into timestamp-delimited "frames" does not work, even if an index is provided for direct access, since an accurate representation of a time interval requires knowledge of events that lie in other frames.

In this simple representation of Fig. 5, it is easy to see that the accurate portrayal of the central frame cannot be done without knowledge of the contents of other frames. Given that states may be nested, the beginning and end of a state that must be shown in a given frame might be many frames away. The straightforward, nonscalable approach, used in our early visualization systems [10,22], was to read the entire file,

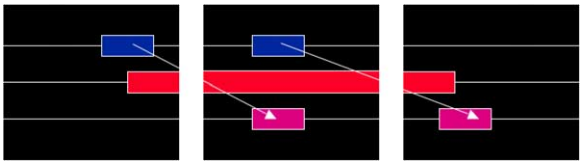

Fig. 5. Three adjacent frames, showing necessity of knowing content of adjacent (and further away) frames.

so that all necessary information was available to the display program. Such files took a long time to load, strained memory limits, and made interactive zooming and scrolling unacceptably sluggish. The key to any scalable approach must be to allow display of a section of the file without reading the whole file.

2.3. Related work

Our first approach (SLOG1 [21]) used frames corresponding to time intervals, with a frame index for scalability. Included in each frame were events from other frames that were "relevant" to the display of this frame. This approach had two problems: first, the two passes (forward and backward) through the CLOG file needed to produce the SLOG1 file were suboptimal, and second, the "shadow" events in a frame sometimes crowded out the frame's proper events. When overcrowding happened, the trace file conversion process took an unacceptably long time.

A recent Dagstuhl seminar provided several other approaches to scalability. The Scalatrace project [17] focuses on mechanisms for compressing the size of the trace files so that assorted analysis tools (such as statistical analysis and replay mechanisms) can still be accurate. The Barcelona group [5] has demonstrated that one apparently obvious scalability problem for large numbers of processes – the limitation on the number of horizontal lines on the display – can be partially overcome for some traces by using a JPEG-like graphical compression on the (virtual) large display needed. Most similar in motivation to our own approach is the Open Trace File format OTF [12], which approaches the "tricky" problem by periodically writing "snapshot" records into the trace file so that the trace can be examined starting at any one of these points. This approach is similar to the one we originally took in SLOG1.

3. The SLOG2 file structure

In this section we describe the SLOG2 file format, how it is created and how it is used.

3.1. The main idea

The goal in SLOG2 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 Fig. 1.

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 Fig. 6. It is related to the R-tree approach described in [1,6].

Perhaps the best way to describe the SLOG2 format for the trace data is to describe the algorithm for computing the tree shown in Fig. 6. 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 states (represented as rectangles in the display) 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). Here we use the half-open interval. The rectangles are sorted by end time, t_e. All events (points in time) are simply placed in the containing leaf node in the tree.

The SLOG file format describes 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 length of the time interval is less than or equal to ΔT_{min}. This value is chosen to work well with the tools that will make use of the SLOG file. Below we describe one way in which ΔT_{min} can be computed. In Section 6, we describe a number of generalizations to this definition. States (rectangles) are placed into the smallest (in time interval) containing node.

In order to display any graphical objects around any chosen time, only a subset of the nodes of the tree must be read. Specifically, in order to display a time t, only the nodes of the tree whose time intervals intersect with this time must be read and displayed. To further reduce the response time seen by users when scrolling forward or backward in time, if t is near either end of an interval, we may choose to read the adjacent interval. In this case, "near in time" could be defined in terms of the expected scrolling behavior.

In this simplest form, the tree is completely balanced. To determine the size of the tree, we need to know the minimum length of time for a leaf node in the tree, ΔT_{min}. Under the assumption that records are uni-

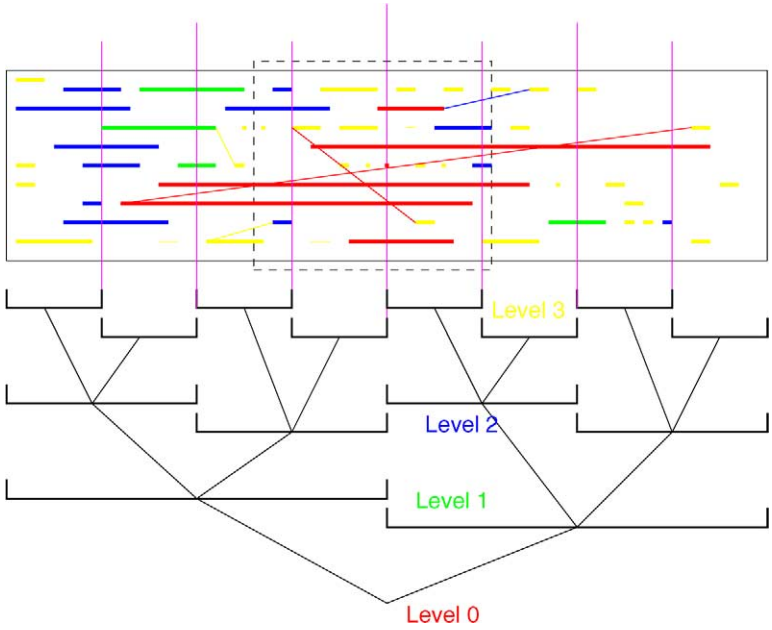

Fig. 6. Bounding boxes for the trace file in Fig. 1. The figure shows the intervals defined as bounding boxes as horizontal brackets, along with the hierarchy of bounding boxes. Colors indicate the assignment of graphical objects to bounding box levels.

formly 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 and that the display will show only a single time interval at any time. Then

$$\Delta T_{\text{min}} = \frac{1}{2} T 2^{-\lfloor \log_2 \frac{n_f}{n_{\text{max}}} \rfloor} \geqslant \frac{1}{2} T \frac{n_{\text{max}}}{n_f}.$$

(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 two leaf nodes.)

If the tree is not perfectly balanced, then ΔT_{min} can be computed in terms of the depth L of the unbalanced tree, as

$$\Delta T_{\text{min}} = \frac{1}{2} T 2^{-L}.$$

Note that in the pathological case of all events within an infinitesimal interval of time, our approach provides no benefit. However, in our experience, events in trace files, while not uniformly distributed, are usually distributed smoothly throughout most of the trace file.

3.2. SLOG2 file format

The bounding-box idea for a trace file and the algorithm to compute the bounding boxes can be applied to many trace file formats. In this section, we briefly describe the SLOG2 file format. This file exploits the post mortem nature of trace files. This allows us to collect data into logical groups, rather than forming it as a stream of records. It also briefly mentions additional data that may be included within the SLOG2 file to aid in analysis or display of performance data. The file format is summarized in Fig. 7.

In the following, we assume that the tree has levels 0 through L. For reference to the algorithm presented in Fig. 8, we provide a correspondence with the notation used in that algorithm, specifically we use R_ℓ as the objects within the relevant time range that are placed at level ℓ in the tree.

header: The file header, containing information on the version of SLOG2, name of the program and the user, and other data about the file.

leaf: Block of data corresponding to the lists of objects R_L, that is, the leaves of the tree.

nonleaf: Block of data corresponding to R_ℓ for $\ell < L$, that is, the interior nodes in the tree.

tree directory: Block of offsets to the beginning of each treenode (both leaf and nonleaf nodes), 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.

postamble: Contains a 4-byte integer indicating the location, relative to the end of the file, of the be-

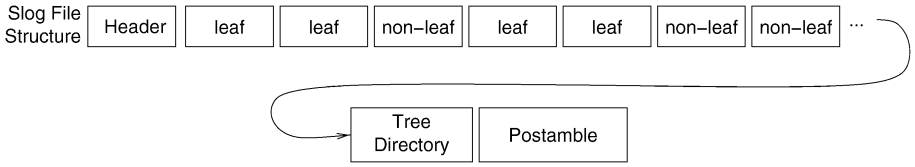

Fig. 7. A simplified block diagram of a complete SLOG2 file. Note that SLOG2 does not define a file format; this block diagram shows one possible organization to illustrate both the contents of the file and a structure that allows the creation of an SLOG2 file in a single pass.

```
for ℓ = 0, 1, ..., L do {
    Set R_ℓ to empty
    Set ΔT_ℓ to [0, 2^{-ℓ} T)
}
Open trace file
while not done {
    read the next state r.
    accumulate any statistics or coordinate mapping data on r
    for ℓ = L, L − 1, ..., 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^{-ℓ} 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 ℓ = L, L − 1, ..., 0 do {
    Write out list R_ℓ. Record the location of R_ℓ in the file in directory D.
}
Write out the directory D.
Write 2^L − 1 (the number of directory entries) as an integer
```

Fig. 8. One-pass algorithm to create an SLOG2 file from a trace file.

ginning of the tree directory. Also contains additional offsets to other data blocks that are part of SLOG2 but are not described in this paper.

The ordering of the blocks is chosen to make it easy to write an SLOG2 file with a single pass. With the exception of the tree nodes, each of these blocks is relatively small.

3.3. Single-pass creation of SLOG2 file

Assume that we have a trace file containing states, sorted by endtime. The trace file may also contain events; these are easily handled, and thus we do not include them in the description of this algorithm. In this section we show how to create the SLOG2 file in a single pass.

Recall that a state represents a single drawable object with a known start and end time and is often drawn as a rectangle. We wish to create an SLOG2 file from the original trace file in a single pass. We assume that most states 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 SLOG2 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 states 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. 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 covers a time interval of length $2^{-\ell}T$.

The algorithm to write an SLOG2 file is shown in Fig. 8.

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 exception 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.

The description of the algorithm considers time intervals and calls them states. In fact, a display program may need to draw a number of different objects, including states (such as the duration of a routine along a timeline), messages (arrows from an event in one timeline to an event in a different timeline), and even polygons (e.g., containing all of the states in a collection of timelines associated with a collective communication operation). All of these can be handled by the SLOG2 file format; to emphasize this, some of the text refers to "drawables" rather than the simpler case of "states".

3.4. Using the SLOG2 file for timeline display

Reading all of the tree nodes for a given interval in an SLOG2 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.

Here we finally see how the SLOG2 format supports responsive interactive zooming and scrolling, while solving the "tricky" part of the problem. The directory at the end of the file allows the display program to seek to and read a *set* of nodes, containing all the intervals relevant to a time interval about a given point in time selected with the mouse. As one scrolls forward or backward at a given zoom level, many – if not most – of the events needed for the display will already be in memory, and only new leaf nodes will need to be read. A limited amount of speculative read-ahead (in both directions) makes scrolling smooth.

After implementing the basic approach described here, we found an additional use for the hierarchical structure of the SLOG2 file. By storing summary data accumulated during the CLOG-to-SLOG2 conversion process in the lower (closer to the root) nodes of the tree, we could present an initial view of this summary data by reading only these nodes and none of the leaf nodes at all. Thus, this view comes up quickly when Jumpshot is started, and presents a summary view of the entire run. It shows "message arrows" representing perhaps thousands of individual messages each and colored blocks representing proportional amounts of time spent in each state (see the left side of Fig. 3). Such a summary view, while lacking in detail, does show overall time distribution to various dominant states as well as the overall communication pattern as it varies over the course of the run. Its primary purpose is to guide the user to where to zoom in. When the user zooms in, which is done by sweeping out an arbitrary extent of time with the mouse), he may encounter a number of levels of summary view before getting down to the individual states and messages in the leaf nodes. Because of the SLOG2 file structure, each zoom operation needs to read only a limited number of nodes of the tree.

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 state category as the third coordinate.

The preceding discussion has assumed that the trace file already contains the necessary states 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 state (exploiting the sequential access to the trace file needed by the algorithm in Fig. 8), preserving the one-pass nature of this algorithm.

4. Experiments

Performance of an interactive program like Jumpshot is primarily a subjective issue. Versions of Jumpshot that read CLOG files were unacceptably slow; the current version, which exploits the features of SLOG2 files, is satisfyingly responsive. Nonetheless, we decided to carry out some experiments in order to quantify the benefits of the file format. We focus on just the time it takes to read those sections of the file requested by a user zoom or scroll operation. This includes, for the SLOG2 case, the time to read the directory and the various nodes of the SLOG2 tree necessary for displaying the relevant time interval.

We compare the time to read a section of trace file stored in the conventional way (ordered strictly by event time), CLOG2 format, with one stored in the SLOG2 format. We profiled FLASH2's sedov3d problem on 16 processes and 700 timesteps to generate a 19 GB CLOG2 trace file, which is then converted to generate a 10 GB SLOG2 file.

In traditional trace display program, the trace events are first assembled into drawable objects that are then fed into the display program in increasing end-time order. The increasing end-time ordering of drawables forces the visualization program to parse the whole trace file whenever the viewport's time range is changed. For example, a long running state starting at the very beginning and ending at the very end of the program will not be seen by the visualization program till the whole trace file is completely parsed. Essentially the response time of the traditional display program is characterized by the time necessary to parse all the trace events into drawable objects. For our 19 GB CLOG2 trace file, the time is 314.5 s, which is too slow for any real-time visualization program.

For the 10 GB SLOG2 file, the first experiment is performed by first zooming to the center of the file (in time) till we see all the real drawables with a viewport size typical of Jumpshot operations. Then we scroll forward in time with the same viewport size till we reach the end of the trace file. The second experiment is similar to the first except we scroll backward in time after the zoom. Plots of the time taken for each zoom or scroll operation vs the start time of the viewport for these experiments are shown in Fig. 9. The most expensive operation in these two experiments is the initial zoom step which takes 140–180 ms. The majority of scroll operations take less than 100 milliseconds. This is over 3000 times faster than with the older CLOG format.

The last experiment we performed aims to show the zooming performance of the SLOG2 format. We first zoom to the center of the trace file as in the previous experiment. Then we jump to the left and then to the right of the center of the trace file with nonoverlapping viewports. The range of jumps increases till it reaches the total duration of the trace file; that is, the last jump of the experiment is from the beginning to the end of the trace file, hence the most expensive operation, about 150 ms. The data is shown in Fig. 10. Most operations take less than 100 ms. The data shows the time that it took to move to the time location indicated on the x-axis from the previous location in time.

Figures 9 and 10 show that the SLOG2 file format allows almost constant access time of drawables of interest by the visualization program, about 100 ms for our 10 GB SLOG2 file. The fast accessing time allows

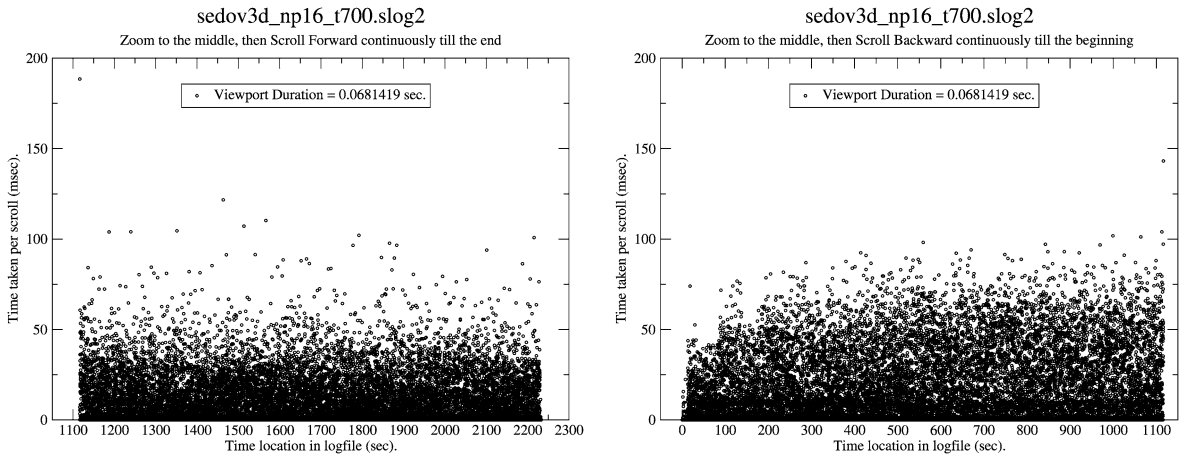

Fig. 9. Plots of time taken for zoom and scroll operations: (left) zoom to the center of the trace file, and then scroll forward till the end of the trace file; (right) zoom to the center of the trace file, and then scroll backward till the end of the trace file.

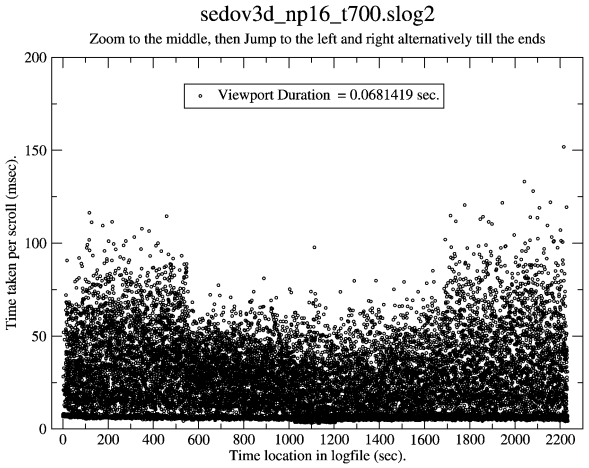

Fig. 10. Zoom to the center of the trace file, and then jump to the left and then right of the center of the trace file with nonoverlapping viewports of same size. The range of the jumps increases till it reaches the total duration of the trace file.

the display program to be responsive in a typical desktop environment.

5. Quantifying data motion

One feature of the simple approach described in Section 3.1 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 situation 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 pathological 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 (nonleaf) bounding boxes can be bounded. If only states are included, the number of objects can often be effectively bounded. However, if drawable objects include connections between send and receive events or nonblocking I/O operations, the number of objects that can cross a particular timeline 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

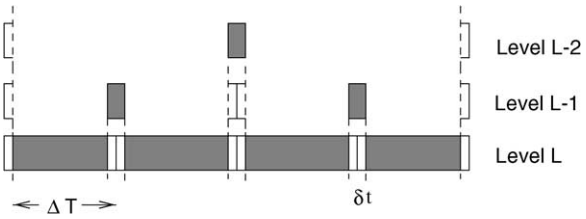

Fig. 11. Any record whose time center falls within the grey area is assigned to the corresponding node. Note that most of the records are assigned to leaf nodes.

algorithm in Fig. 8, where the time intervals for each node on a level are disjoint (nonoverlapping). 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 $N\delta t/\Delta T_{\text{min}}$ will be placed in the leaf nodes. Of the remaining records, $N\delta t/(2\Delta T_{\text{min}})$ will be placed in the nodes at the next level, $N\delta t/(4\Delta T_{\text{min}})$ in the next, down to level 0. This approach is illustrated in Fig. 11, where level 0 is at the top of the figure (to maintain the usual convention of trees growing down in computer science).

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

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

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

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

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 $N\delta t \leqslant T$, 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 than the number of available pixels (in the vertical dimension) for the graphical representation to be meaningful. This requirement also provides a bound on the number of overlapping objects in a *useful* SLOG2 file.

If too many drawable objects of short duration cross the boundaries between tree nodes, however, 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 strategy handles the pathological case mentioned above.

6. Extensions and refinements

Here we describe some more detailed variations and alternative uses of the main SLOG2 idea.

6.1. Real-time file generation

Instead of creating an SLOG2 file from an existing trace file (as in Fig. 4), we can create the SLOG2 file directly from the program that is creating the logfile. If the trace file is created at one time when the program is exiting, then there is no problem. However, trace file tools commonly bound the amount of memory used to store trace file data; the trace file is appended to as the internal buffer fills up. Hence, the total time interval T for the entire run is not known in advance. In this case, the algorithm in Fig. 8 can still be used with a few changes. Specifically, the total number of levels is not set in advance; instead, 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 adds 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*.

6.2. Variable leaf sizes

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. In this case, instead of preselecting a value of $\Delta T_{\min}$, the length of time for each leaf interval is determined as the records are read.

6.3. File compression

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 SLOG2 file is (often) generated after the run of a program, we can use a different approach based on applying data compression to each tree node as a block. The tree node header indicates which type of compression has been used on the rest of the tree node. 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, [7, 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.

6.4. Annotating existing files

An alternative to writing an SLOG2 file from trace data is to simply *annotate* an existing trace file with the additional information required to define the nonleaf nodes in the tree of bounding boxes. The same one-pass algorithm can be used to create this annotation file, with the difference that instead of writing out the leaf nodes of the tree, a record is written that points to the range of bytes in the original trace file containing the events for that leaf node. A display or analysis program that is using an annotation file must then filter the data from the original trace file, since that block of data will contain events that may belong to other (nonleaf) nodes in the tree. In this way, an SLOG2 annotation may be combined with any other trace file.

7. Summary

We have described a hierarchical file format, SLOG2, that enables a full-featured trace file display program to remain interactively responsive in the face of large trace files. We have provided the algorithm for creating the file and presented some analysis to support the decisions that we made. Our measurements show that interactive visualization of a multigigabyte trace file is possible with the SLOG2 format. We have also described a number of options and extensions to the implementation described here.

Acknowledgments

This work was supported in part by the US Department of Energy Contract #B523820 to the ASC/Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago and in part by the Mathematical, Information and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, US Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] R tree, http://en.wikipedia.org/wiki/R-tree.

[2] J. Chassin de Kergommeaux and B. Stein, Pajé, an extensible environment for visualizing multi-threaded programs executions, in: *Proceedings of Euro-Par 2000*, A. Bode et al., eds, LNCS, Vol. 1900, Springer-Verlag, Berlin, 2000, pp. 133–140.

[3] J. Chassin de Kergommeaux, B. Stein and P.E. Bernard, Pajé, an interactive visualization tool for tuning multi-threaded parallel applications, *Parallel Computing* **26** (2000), 1253–1274.

[4] TimeScan Multiprocess Event Analyzer, http://www.etnus.com/products/timescan.

[5] J. Gemenez, Are you sure that tracing is not scalable?, in: *Dagstuhl Seminar, "Code Instrumentation and Modeling for Performance Analysis"*, http://kathrin.dagstuhl.de/07341, August 2007.

[6] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: *SIGMOD'84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ACM Press, New York, NY, USA, 1984, pp. 47–57.

[7] D. Hankerson, G.A. Harris and P.D. Johnson Jr., *Introduction to Information Theory and Data Compression*, CRC Press, Boca Raton, FL, 1998.

[8] M.T. Heath, Recent developments and case studies in performance visualization using ParaGraph, in: *Performance Measurement and Visualization of Parallel Systems*, G. Haring and G. Kotsis, eds, Elsevier Science Publishers, Amsterdam, 1993, pp. 175–200.

[9] M.T. Heath and J.E. Finger, Paragraph: A performance visualization tool for MPI, http://www.csar.uiuc.edu/software/paragraph.

[10] V. Herrarte and E. Lusk, Studying parallel program behavior with upshot, Technical Report ANL–91/15, Argonne National Laboratory, 1991.

[11] IBM, *IBM Parallel Environment for AIX*, Vol. 2 (Chapter 3), IBM, 1998; http://www.s390.ibm.com/bookmgr-cgi/bookmgr.cmd/books/PEOP2240/CONTENTS.

[12] H. Jagode, Random access to event traces with OTF, August 2007; http://kathrin.dagstuhl.de/07341/Materials2.

[13] E. Karrels and E. Lusk, Performance analysis of MPI programs, in: *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing*, J. Dongarra and B. Tourancheau, eds, SIAM Publications, Philadelphia, PA, 1994, pp. 195–200.

[14] E. Lusk, Performance visualization for parallel programs, *Theoretica Chimica Acta* **84** (1993), 377–384.

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

[16] A.D. Malony, D.A. Reed and H.A.G. Wijshoff, Performance measurement intrusion and perturbation analysis, *IEEE Transactions on Parallel and Distributed Systems* **3**(4) (1992), 433–450.

[17] F. Mueller, M. Noeth, P. Ratn, M. Schulz and B.R. de Supinski, Scalable compression and replay of communication traces, August 2007; http://kathrin.dagstuhl.de/07341/Materials2.

[18] Vampir 2.0 – Visualization and analysis of MPI programs, http://www.pallas.de/pages/vampir.htm.

[19] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz and L.F. Tavera, Scalable performance analysis: The Pablo performance analysis environment, in: *Proceedings of the Scalable Parallel Libraries Conference*, A. Skjellum, ed., IEEE Computer Society, Los Alamitos, CA, 1993.

[20] G. Tomas and C.W. Ueberhuber, *Visualization of Scientific Parallel Programs*, LNCS, Vol. 771, Springer-Verlag, Berlin, 1994.

[21] C.E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E.L. Lusk and W. Gropp, From trace generation to visualization: A performance framework for distributed parallel systems. in: *Proceedings of SC2000*, Los Alamitos, CA, 2000.

[22] O. Zaki, E. Lusk, W. Gropp and D. Swider, Toward scalable performance visualization with Jumpshot, *High Performance Computing Applications* **13**(2) (1999), 277–288.