Large Event Traces in Parallel Performance Analysis

Felix Wolf[1], Felix Freitag[2], Bernd Mohr[1], Shirley Moore[3], Brian Wylie[1]

[1]Forschungszentrum Jülich, ZAM
52425 Jülich, Germany
{f.wolf, b.mohr, b.wylie}@fz-juelich.de

[2]Universitat Politécnica de Catalunya, Computer Architecture Dept.
08034 Barcelona, Spain
felix@ac.upc.es

[3]University of Tennessee, Innovative Computing Laboratory
Knoxville, TN 37996, USA
shirley@cs.utk.edu

Abstract:
A powerful and widely-used method for analyzing the performance behavior of parallel programs is event tracing. When an application is traced, performance-relevant events, such as entering functions or sending messages, are recorded at runtime and analyzed post-mortem to identify and potentially remove performance problems. While event tracing enables the detection of performance problems at a high level of detail, growing trace-file size often constrains its scalability on large-scale systems and complicates management, analysis, and visualization of trace data. In this article, we survey current approaches to handle large traces and classify them according to the primary issues they address and the primary benefits they offer.

Keywords: parallel computing, performance analysis, event tracing, scalability.

1 Introduction

Event tracing is a powerful and widely-used method for analyzing the performance of parallel programs. In the context of developing parallel programs, tracing is especially effective for observing the interactions between different processes or threads that occur during communication or synchronization operations and to analyze the way concurrent activities influence each other's performance.

Traditionally, developers of parallel programs use tracing tools, such as Vampir [NWHS96], to visualize the program behavior along the time axis in the style of a Gantt chart (Figure 1), where local activities are represented as boxes with a distinct color. Interactions between processes are indicated by arrows or polygons to illustrate the exchange of messages or the involvement in a collective operation, respectively.

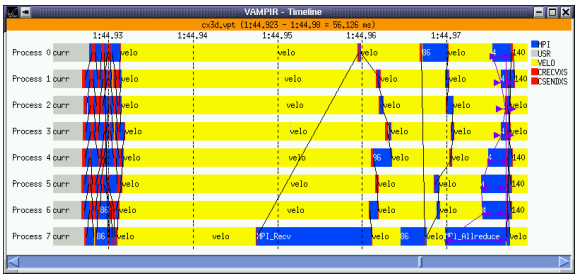

Figure 1: Vampir time-line visualization of an MPI application.

To record a trace file, the application is usually instrumented, that is, extra code is inserted at various levels that intercepts the desired events and generates the appropriate trace records. The events monitored typically include entering and leaving code regions as well as communication and synchronization events happening inside these regions. Memory hierarchy events comprise another important event type which is predominantly used in cache-analysis tools.

Trace records are kept in a memory buffer and written to a file after program termination or upon buffer overflow. Recording communication and synchronization events is often accomplished by interposing wrappers between application and communication library during the link step. While the program is running, each process or thread generates a local trace file, which is merged into a global trace after program termination.

Scalability, as is described in Section 2, is a major limitation of trace-based performance analysis. Due to its ability to highlight complex performance phenomena, however, trace-based performance analysis will continue to be needed to achieve an efficient utilization of massively-parallel systems and that research directed towards improving the scalability of this diagnosis technique is a worthwhile undertaking.

In this article, we give an overview of existing approaches to limit the amount of trace data needed or to efficiently handle larger event traces if they cannot be avoided. We start in Section 2 with a discussion of technical issues limiting the scalability of trace-based performance analysis. In Section 3 we distinguish situations leading to large traces. The actual survey of approaches along with our classification is presented in Section 4, followed by our conclusion in Section 5.

2 Scalability Issues

Although event tracing is a powerful performance-diagnosis technique, is has known limitations on large-scale systems. This article focuses on problems related to trace-file size.

Another set of problems is related to synchronization of event timings. Since the comparison of event timings across the processes of a parallel program is an important element of trace-based performance analysis, the absence of globally synchronized clocks may

adversely affect the accuracy and consistency of event measurements. This problem has been addressed by hardware solutions, such as the BlueGene/L global barrier and interrupt network, runtime measurements with linear interpolation [WM03], and off-line correction based on logical clocks [Rab97]. Although still an open research area, this issue is beyond the scope of this paper.

The amount of trace data generated poses a problem for (i) management, (ii) visualization, and (iii) analysis of trace data. Note that often these three aspects cannot be clearly separated because one may act as a tool to achieve the other, for example, when analysis occurs through visualization.

The size of a trace file may easily exceed the user or disk quota or the operating-system imposed file-size limit of 2 GB common on 32-bit platforms. For SciDAC application Gyro [CW03], we have obtained about 2.9 MB of trace data per process for a varying number of processes - even after applying a relatively selective instrumentation scheme. Extrapolating this number to 10,000 processes would result in more than 20 GB of data.

However, even if the trace data are divided into multiple files, as supported by Intel's STF trace format, moving and archiving large amounts of trace data can be tedious and cumbersome. Since a typical performance analysis cycle usually involves several experiments to adequately reflect different execution configurations, input data sets, and perhaps program versions, the amount of data becomes multiplied by a larger factor.

Building robust and efficient end-user tools to collect, analyze, and display large amounts of trace data is a very challenging task. Large memory requirements often cause a significant slow down or - even worse - place practical constraints on what can be done at all. Moreover, if the trace data of a single program run are distributed across multiple files, for example, before merging local files into a single global file, trace processing as it occurs during the merge step may require a large number of files to be open simultaneously, creating a potential conflict with given operating-system limits.

Even if the data management problem can be solved, the analysis itself can still be very time consuming, especially if it is performed without or with only little automatic support. On the other hand, the iterative nature of many applications cause trace data to be highly redundant as the same code is repeated many times with nearly identical computations (e.g., in loops). Thus, compression techniques can be useful for reducing the amount of data to be stored and analyzed.

In the next section, we discuss why event traces become large as a foundation for our later classification of approaches to improve scalability.

3 Reasons for Large Traces

The reasons for large traces can be roughly divided into five categories:

Number of processes/threads Since this number is equal to the number of time-lines in a time-line diagram, it is often referred to as the *width* of an event trace (as opposed

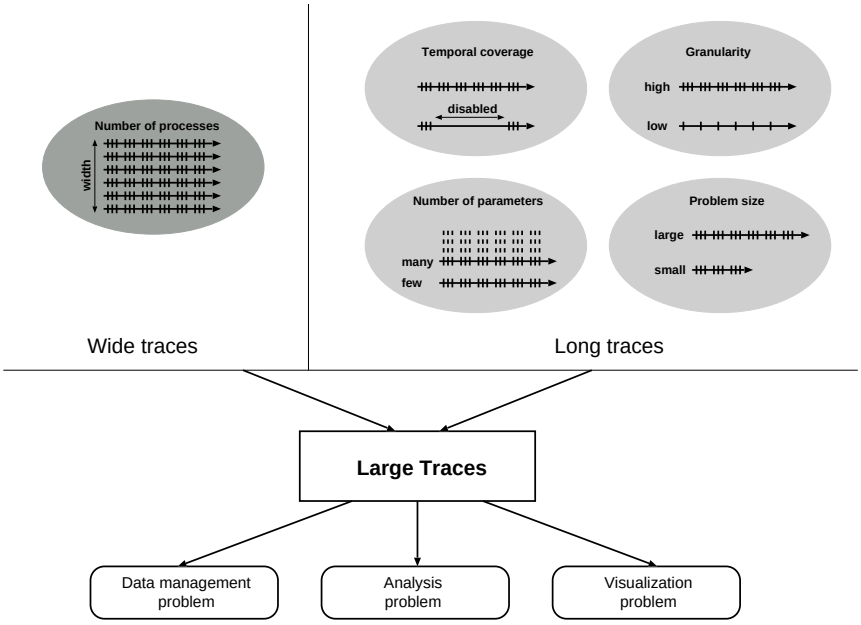

Figure 2: Reasons for large traces and the problems they cause.

to the *length*, which represents the number of events per process). Because the total number of communication and synchronization events usually grows with the number of processes, the width influences both the total amount of data as well as the total number of local trace files that need to be handled.

Temporal coverage The intervals to be traced need not cover the entire execution. It is obvious that restricting tracing to smaller intervals can substantially decrease the amount of trace data.

Granularity How many events are recorded during a given interval depends on the frequency at which events are generated. This is typically related to the granularity of measurements, that is, the level of detail (e.g., function, block, or statement level) captured through tracing. The reader should note that a high frequency can also increase perturbation and alter performance behavior.

Number of event parameters The parameters recorded as part of an event typically include a process identifier, a time stamp, and a type identifier. In addition, there may be one or more type-specific parameters. In parallel performance analysis the number of parameters rarely exceeds a few unless hardware counter readings, of which there may be many, are added to the event record.

Problem size This factor considers the number of performance-relevant events as a result of the algorithm applied to a certain input problem. A typical example is the number

of iterations performed to arrive at a solution, which can prolong execution time and increase the number of events to be traced.

Figure 2 summarizes causes and effects of large traces. Note that all reasons except for the number of processes fall under the category long traces. However, it should be noted that growing problem sizes often demand a higher degree of parallelism.

4 Approaches to Improve Scalability

In this section, we discuss several approaches to either avoid or to handle large traces. At the end, we classify the different methods by causes and effects of the scalability problem they address.

Frame-based data format. To allow for efficient zooming and scrolling, traditional trace-visualization tools require the event trace to reside in main memory. As traces grow bigger, methods are needed to either efficiently access trace data from files or to create a more compact main-memory representation.

An approach targeting the first option has been developed by Wu et al. [WBS+00]. They have introduced a trace-file format named SLOG that supports scalable visualization in the sense that CPU-time and memory requirements depend only on the number of graphical objects to be displayed and neither on the total amount of trace data nor on a particular interval chosen for display.[1]

If the user wants to display only a section of an event trace, the SLOG format allows the viewer to read only the necessary portions of the file. The trace file is divided into frames (Figure 3) representing different intervals of program execution. To complete the visualization of an interval frame, each frame includes so-called pseudo records that contain state information from outside the interval, such as message events required to draw message arrows beginning or ending outside the displayed section. Linked frame directories enable rapid access to other time intervals even if they are located far into the run.

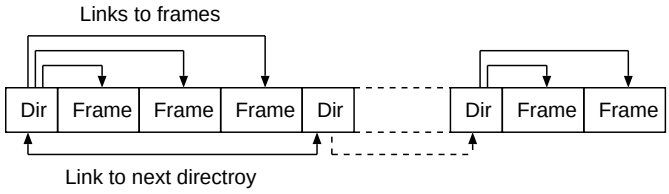

Figure 3: The SLOG trace-file structure is divided into frames representing separate intervals.

Work presented in [CGL00] further improves visual performance by arranging drawable objects into a binary tree of bounding boxes, which also eliminates the need for pseudo records and provides better support for drawing coarse-grained previews.

[1] A similar approach is supported by the STF format.

Periodicity detection. As pointed our earlier, many scientific applications exhibit a highly repetitive behavior as a result of performing loop iterations of almost identical calculations. If these iterations perform in the same manner, tracing only a few instances is often sufficient to capture the entire application behavior, that is, tracing can be restricted to representative intervals, resulting in significantly reduced trace-file length.

Freitag et al. [FCL02] have developed a dynamic mechanism for OpenMP applications that is able to detect repetitive behavior at run time and to disable tracing after recording a representative number of iterations. Once the periodic behavior ends, tracing is resumed until the next periodic section is reached. In this way, the trace-file size obtained for the NAS LU benchmark was reduced by 95 %. Periodicity is detected using a distance metric applied to the stream of function identifiers representing the sequence of function entries and exits:

$$d(m) = sign \sum_{i=0}^{N-1} |x(i) - x(i-m)|, \quad 0 < m < M \wedge M \leq N \quad .$$

The formula computes the vector distance between vector $x(i)$ and the same vector shifted by m samples. N is the size of the data window, M is the maximum period length that can be detected. If the equation returns zero for a given m, the data window contains an identical periodic pattern with periodicity m. The evaluation of the distance is based on an efficient algorithm that requires $O(M)$ calculations per event. Compared to the original tracing mechanism, which added about 1-3% of overhead to the execution time, periodicity detection increased the overhead to only 3-6%.

Call-graph compression. If an event trace is stored in main memory, it is usually organized as a linear list of event records in chronological order. Knüpfer and Nagel [KN05] proposed an alternate data structure named *Complete Call Graph* (CCG) which arranges event records in a tree resembling the program's function call hierarchy, where each node represents a function call. In contrast to linear lists, time stamps appear transformed into the durations that lie between two subsequent events. Each local trace is transformed into a separate CCG.

The main advantage over linear lists is the ability to merge equal or similar nodes with the intent of "shortening" traces. This is achieved by eliminating redundancy along the time axis created as result of iterative behavior - similar in spirit to the previously discussed on-line periodicity detection only that it is performed off-line.

Compressed CCGs (cCCGs) are created from regular CCGs by applying either lossless or lossy compression. It is important to note that this occurs entirely transparent to read access, in that a later decompression is not required. The deviation arising from lossy compression can be controlled both in terms of the absolute maximum deviation of individual time stamps and in terms of the relative percentage that the durations between subsequent events are changed.

The computational effort required for CCG construction is $O(n)$ with n being the total number of events. The overall complexity of CCG compression is $O(N * m)$ with N being the total number of nodes in the CCG and m a rather small data-dependent factor. Experimental results showed memory compression ratios between 2 to 8 for lossless compression

and up to 44 for lossy compression with midrange deviation bounds (1000 clock ticks and 50 % interval deviation).

CCGs not only save memory, they can also speed up analysis. Queries, such as summary statistics, are most efficiently computed in a recursive manner along the CCG tree structure. *Caching* of intermediate results at graph nodes can significantly accelerate subsequent queries. Since in compressed trees redundant nodes have been merged, cached results can be reused even within the same query.

Distributed analysis. Whereas the previous two approaches primarily address long traces, Brunst and Nagel [BN03] have designed and prototyped a distributed performance analysis service suitable to access and evaluate, in particular, wide traces collected on machines with a large number of CPUs. The service is called Vampir Next Generation (VNG) and essentially constitutes a scalable version of the popular Vampir event-trace browser.

VNG is based on the principle of keeping performance data close to the location where they were created and of exploiting distributed memory for the analysis. It consists of a parallel analysis server and a potentially remote visualization client. The server is submitted post-mortem as a separate parallel job usually on the machine where the trace data were created. The client runs on the user workstation and interacts with the server via a network.

The parallelization is based on MPI combined with pthreads and follows a master-worker scheme. A master process handles incoming client requests by partitioning the work and delegating the resulting subtasks to the worker processes. These are responsible for storage and analysis of different subsets of the overall trace data, which are distributed across multiple files. To support multiple client sessions, each server process may employ several session threads. Similar to the SLOG approach, VNG limits the amount of data maintained by the visualization client to a volume that is independent of the amount of traced event data. Benefits of the distributed architecture have been demonstrated most notably for summary charts and load operations with super-linear speed-ups of 60 and 40, respectively, measured for $16 + 1$ processes. To further improve the support of long traces, VNG has been successfully combined with CCG compression.

Automatic pattern search. The approaches covered so far have been devised mainly with graphical trace browsers in mind. Either the amount of trace data has been reduced and/or the access to it has become more efficient. However, from a user's perspective nothing has changed, the general way of looking at the data is still the same: zooming in and out until a phenomenon of interest has been found. Especially in view of large-scale systems creating a need to deal with larger traces, this process can be time consuming and - given the size of the search space - complete coverage is hard to achieve.

An approach based on automatic pattern search, which was developed by Wolf and Mohr [WM03] and implemented in the KOJAK tool, tries to combine the expressiveness of trace data with the compactness of profile data. KOJAK scans trace files off-line for execution patterns representing inefficient behavior, such as communication and synchronization delays, thus relieving the user from the burden of searching large amounts of trace data manually. Since KOJAK distinguishes patterns of different shape, the performance behavior is automatically classified by bottleneck type. Once a match has been found, the detected

pattern instance is gauged to quantify its performance impact in terms of the time lost due to its occurrence. The measurements are accumulated and mapped onto the call tree and the hierarchical structure of the parallel system. The resulting three-dimensional representation of performance behavior owes its compactness the fact that the time dimension (i.e., length) of the trace has been made implicit by aggregation of quantified pattern matches. The automatic analysis has been optimized by exploiting the specialization hierarchy in which the different patterns are arranged. In addition to speedups between 2.5 and 13, this strategy has also reduced the amount of code needed to describe the patterns.

Topological analysis. The integration of process topologies into the KOJAK framework [BSW+05] has improved the analysis of wide traces by visually grouping processes according to their virtual or physical topological characteristics and by helping identify macro-patterns that put primitive pattern instance into a larger context covering the entire topology.

Holistic analysis. In order to assess single-node performance in addition to the time-based patterns focusing primarily on parallel interaction, the KOJAK framework allows the user to record a large variety of hardware-counter readings including cache misses, TLB misses, and floating-point operations as part of certain event records. As pointed out in Section 3, this can easily lead to a significant trace-file enlargements, especially when data from a large number of counters are recorded to gain insights from comparing counter values across various functional units. To avoid trace-file enlargement proportional to the number of recorded counters, Wylie et al. [WMW05] replace a single experiment by multiple experiments conducted under similar conditions, each time recording a different subset of the desired counters. The resulting traces are analyzed separately and the results merged into a single integrated "holistic" picture of performance behavior.

Granularity reduction. Indiscriminate instrumentation of user functions can easily lead to significant trace-file enlargement and perturbation, often caused by short but frequently executed user-function calls without any communication. Since trace analysis focuses mostly on communication and synchronization, recording functions not involved in such operations rarely contribute to the analysis. To keep trace-file size within manageable bounds, Moore et al. [MWD+05] devised a semi-automatic two-step process to prevent calls to these functions from being recorded: First, a call path profile is generated using the TAU tool [MS03]. Next, a script automatically extracts all function names directly or indirectly involved in communication or synchronization calls and generates a so-called TAU include list used to restrict instrumentation to those functions only on subsequent runs. Using this method, the trace-file size for the aforementioned Gyro code could be reduced from more than 100 GB (estimated) to less than 100 MB. The latest version of TAU also includes an on-line mechanism to filter out function calls that are not ancestors of MPI calls.

Statistical analysis. Aguilera et al. [ATTW06] apply multivariate statistical techniques to event traces with the aim of isolating processes that might be experiencing communication problems, thus clearly addressing wide traces. Their methodology follows a sequence

of five steps: (i) extract communication data from the trace file, (ii) summarize extracted communication information, (iii) create a distance matrix using data from the previous step, (iv) perform hierarchical clustering, and finally (v) identify process pairs of interest for a more in-depth analysis.

Table 1: Classification of approaches to improve the scalability of trace analysis.

Approach	Primary reason addressed	Primary benefit
Frame-based format	Long traces	Speeds up visualization
Periodicity detection	Temporal coverage	Reduces trace to relevant intervals
Call-graph compression	Temporal coverage	Reduces trace data by averaging similar behavior
Distributed analysis	Number of processes	Speeds up analysis and visualization
Automatic pattern search	Long traces	Speeds up analysis
Topological analysis	Number of processes	Groups processes in analysis and visualization
Holistic analysis	Number of parameters	Distributes parameters across several experiments
Granularity reduction	Granularity	Reduces trace to relevant events
Statistical analysis	Number of processes	Speeds up analysis

To summarize our survey, Table 1 classifies the different methods by the primary reason addressed (Figure 2) and the primary symptom cured.

5 Conclusion

We have discussed several recent approaches to improve trace-analysis scalability and classified them by the primary causes and effects they address (Table 1). Although each constitutes an important improvement, none of them in isolation is powerful enough to meet the extreme scalability requirements of today's supercomputers and applications. However, we notice that the above approaches can be seen as largely orthogonal in the sense that they can be effectively combined to facilitate a higher degree of scalability. For example, as already anticipated by combining VNG with cCCGs, the VNG concept of analyzing trace data in parallel could be extended to the automatic pattern search and enhanced with periodicity detection or call-graph compression.

Acknowledgements This work was partially supported by the Spanish Ministry of Science and Technology under Contract TIN2004-07739-C02-01. We would also like to thank Holger Brunst and Andreas Knüpfer for helping improve our understanding of the VNG and cCCG approaches.

References

[ATTW06] G. Aguilera, P. J. Teller, M. Taufer, and F. Wolf. A Systematic Multi-step Methodology for Performance Analysis of Communication Traces of Distributed Applications based

on Hierarchical Clustering. In *Proc. of the 5th International Workshop on Performance Modeling, Evaluation, and Organization of Parallel and Distributed Systems (PMEO-PDS)*, Rhodes, Greece, April 2006. Accepted for publication.

[BN03] H. Brunst and W. E. Nagel. Scalable Performance Analysis of Parallel Systems: Concepts and Experiences. In *Proc. of the Parallel Computing Conference (ParCo)*, Dresden, Germany, 2003.

[BSW+05] N. Bhatia, F. Song, F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Automatic Experimental Analysis of Communication Patterns in Virtual Topologies. In *Proc. of the International Conference on Parallel Processing (ICPP)*, Oslo, Norway, June 2005. IEEE Society.

[CGL00] A. Chan, W. Gropp, and E. Lusk. Scalable Log Files for Parallel Program Trace Data (DRAFT). Technical report, Argonne National Laboratory, 2000.

[CW03] J. Candy and R. Waltz. An Eulerian gyrokinetic Maxwell solver. *J. Comput. Phys.*, 186:545–581, 2003.

[FCL02] F. Freitag, J. Caubet, and J. Labarta. On the Scalability of Tracing Mechanisms. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, volume 2400 of *Lecture Notes in Computer Science*, Paderborn, Germany, August 2002. Springer.

[KN05] A. Knüpfer and W. E. Nagel. Construction and Compression of Complete Call Graphs for Post-Mortem Program Trace Analysis. In *Proc. of the International Conference on Parallel Processing (ICPP)*, pages 165–172, Oslo, Norway, June 2005. IEEE Society.

[MS03] A. D. Malony and S Shende. Performance Technology for Complex Parallel and Distributed Systems. In P. Kacsuk and G. Kotsis, editors, *Quality of Parallel and Distributed Programs and Systems*, pages 25–41. Nova Science Publishers, Inc., New York, 2003.

[MWD+05] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr. A Scalable Approach to MPI Application Performance Analysis. In *Proc. of the 12th European Parallel Virtual Machine and Message Passing Interface Conference (EUROPVMMPI)*, Sorrento, Italy, September 2005. Springer.

[NWHS96] W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 63, XII(1):69–80, 1996.

[Rab97] R. Rabenseifner. The Controlled Logical Clock - a Global Time for Trace Based Software Monitoring of Parallel Applications in Workstation Clusters. In *Proc. of the 5th EUROMICRO Workshop on Parallel and Distributed Processing (PDP)*, pages 477–484, London, UK, January 1997.

[WBS+00] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In *Proc. of the Supercomputing Conference (SC2000)*, Dallas, TX, November 2000.

[WM03] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003. Special Issue "Evolutions in parallel distributed and network-based processing".

[WMW05] B. Wylie, B. Mohr, and F. Wolf. Holistic hardware counter performance analysis of parallel programs. In *Proc. of Parallel Computing 2005 (ParCo 2005)*, Malaga, Spain, September 2005.