A Parallel Trace-Data Interface for Scalable Performance Analysis

Markus Geimer[1], Felix Wolf[1,2], Andreas Knüpfer[3],
Bernd Mohr[1], and Brian J. N. Wylie[1]

[1] John von Neumann Institute for Computing (NIC)
Forschungszentrum Jülich, 52425 Jülich, Germany
{m.geimer, f.wolf, b.mohr, b.wylie}@fz-juelich.de

[2] Department of Computer Science
RWTH Aachen University, 52056 Aachen, Germany

[3] Center for Information Services and High Performance Computing (ZIH)
Dresden University of Technology, 01062 Dresden, Germany
andreas.knuepfer@tu-dresden.de

Abstract Automatic trace analysis is an effective method of identifying complex performance phenomena in parallel applications. To simplify the development of complex trace-analysis algorithms, the EARL library interface offers high-level access to individual events contained in a global trace file. However, as the size of parallel systems grows further and the number of processors used by individual applications is continuously raised, the traditional approach of analyzing a single global trace file becomes increasingly constrained by the large number of events. To enable scalable trace analysis, we present a new design of the aforementioned EARL interface that accesses multiple local trace files in parallel while offering means to conveniently exchange events between processes. This article describes the modified view of the trace data as well as related programming abstractions provided by the new PEARL library interface and discusses its application in performance analysis.

1 Introduction

Event tracing is a well-accepted technique for post-mortem performance analysis of parallel applications. Time-stamped events, such as entering a function or sending a message, are recorded at runtime and analyzed afterwards with the help of software tools. For example, graphical trace browsers, such as VAMPIR [1] or PARAVER [2], provide a zoomable time-line display, allowing a manual, fine-grained investigation of parallel performance behavior.

However, regarding the large amounts of data usually generated, automatic off-line trace analyzers, such as EXPERT from the KOJAK toolset [3,4], can provide the user with relevant information more quickly by automatically searching traces for complex patterns of inefficient behavior and quantifying their significance. In addition to usually being faster than a manual analysis performed using the aforementioned trace browsers, this approach is also guaranteed to cover the entire event trace and not to miss any pattern instances.

To simplify the analysis logic incorporated in EXPERT, it has been designed on top of EARL [5], a high-level interface to access individual events from a single global trace file. As opposed to a low-level interface that allows reading individual event records only in a sequential manner, EARL offers random access to individual events. Not to restrict trace-file size, EARL assumes access locality allowing it to buffer the context of recent accesses in main memory while reading events outside this context from file. In addition, to support the identification of pattern constituents, EARL provides a set of abstractions representing execution state information at the time of a given event as well as links between related events, such as corresponding enter and exit events for function instances.

Unfortunately, sequentially analyzing a single and potentially large global trace file does not scale well to applications running on thousands of processors. Even if access locality is exploited as described above, the amount of main memory might not be sufficient to store the current working set of events. In addition, the preceeding step of merging local event trace data generated by individual processes into a global trace file is very time-consuming. Moreover, the amount of trace data might not even fit into a single file, which already suggests to perform the analysis in a more distributed fashion.

To enable scalable trace analysis for modern large-scale systems and applications running on them, we have designed a parallel trace-data interface PEARL as a building block for parallel trace analysis algorithms and tools. In this article, we describe the modified view of the trace data in combination with programming abstractions representing this view. We start our discussion with a review of related work in Section 2, followed by a description of the serial interface in Section 3. In Section 4, we detail the programming abstractions offered by the new parallel interface, before presenting the intended usage as a framework for implementing automatic parallel trace analysis in Section 5. Finally, we conclude the paper and outline some further improvements in Section 6.

2 Related Work

In [6], Wolf et al. review a number of approaches addressing scalable trace analysis. The frame-based SLOG trace-data format [7] supports scalable visualization, whereas dynamic periodicity detection in OpenMP applications [8] prevents redundant performance behavior from being recorded in the first place. Important to our approach has been the distributed trace analysis and visualization tool VAMPIR Server [9], which already provides parallel trace access mechanisms, albeit targeting a "serial" human client in front of a graphical trace browser as opposed to fully automatic and parallel trace analysis. Miller et al. have used a distributed algorithm on multiple local trace data sets [10] to calculate the critical path, which identifies parts of the program responsible for its length of execution.

Unlike common linear storage schemes for event trace data, the tree-based main memory data structure called cCCG [11] allows potentially lossy compression of trace data while observing previously specified deviation bounds. Since we are considering to use CCCGs as an alternate base data structure for our trace-data interface, the parallel programming abstractions described in this paper are designed in such a way that the underlying data structure can be easily changed when the library is compiled.

3 Serial Programming Interface: EARL

EARL (Event Analysis and Recognition Library) is a C++ class library that offers a high-level interface to access event traces of MPI, OpenMP, or SHMEM applications. In the context of EARL, an event trace is stored in a single global trace file that includes events from all processes or threads in chronological order. The user is given random access to individual events allowing the retrieval of distinct events by their index within the chronologically sorted sequence. Loops iterating over the entire trace can be easily implemented by querying the total number of events beforehand.

In addition, EARL provides execution state information at the time of a given event in the form of event sets describing a particular aspect of this state. The state being calculated is either local or global. Local state always refers to a single process or thread, whereas global state may encompass multiple processes or threads. Local state information provided by EARL includes the call stack in form of the enter events of currently active region instances; global state information includes the set of messages currently in transit represented by their respective send events, completed collective operations represented by their respective exit events, and the global call tree derived from the different call stacks, as it evolves over time. Based on this state information, EARL also provides links between related events, which are called pointer attributes. Pointer attributes can also be divided into local and global attributes. There is one local attribute pointing to the enter event of the currently active region instance and allowing traversal of the call stack. Several global attributes support functions, such as locating the send event corresponding to a given receive event, uniquely identifying call paths, traversing the global call tree, or following the ownership history of OpenMP locks between threads.

The intended trace analysis process supported by EARL is a sequential traversal of the event trace from beginning to end. As the analysis progresses, EARL updates the execution-state information and calculates pointer attributes for the most recent event being read, which always point backwards to avoid a costly look-ahead. To make the trace analysis process more efficient, EARL buffers the context of the current event so that events within this context can be directly accessed from main memory. This context includes the last *n* events (i.e., the history), including the entire related execution-state information.

To avoid re-reading the trace file from the very beginning in cases where an event outside the context is requested, EARL additionally stores the complete execution state information at regular intervals in so-called bookmarks. The history size as well as the bookmark distance can be flexibly configured, however, since these parameters have a significant performance impact with respect to memory consumption and the number of required file accesses and, in addition, these effects are highly application dependent, finding an optimal set of parameters is a non-trivial task. For more details about EARL, the interested reader may refer to the user manual [12].

4 Parallel Access to Trace Data: PEARL

In this section, our new parallel programming interface for accessing event trace data is presented. Before going into the details of the programming abstractions provided, we start with an outline of the overall design and show how it differs from the previous approach described above.

4.1 Design Overview

Similar to EARL, our new parallel trace data access interface, which we call PEARL, is also implemented as a C++ class library. However, we no longer assume a single global trace file, as was the case for the serial interface. Instead, PEARL operates on multiple process-local trace files.

For simplicity, our initial implementation of the PEARL library focuses only on single-threaded MPI-1 applications. However, during the entire design it was taken into account that we plan to extend this approach to alternate parallel programming models, such as OpenMP or MPI-2. Therefore, the PEARL library interface is subdivided into a generic part, which is independent of the programming model used, and an MPI-specific part, implemented by deriving specialized subclasses from the generic interface. Once support for multi-threaded applications has been added, PEARL will access one local trace file per thread.

The benefit behind the approach of using multiple local trace files instead of a single global file is twofold. First, it avoids the time-consuming step of merging the process-local trace files generated by the measurement system into a single global file. And second, it allows us to effectively exploit the distributed memory and parallel processing capabilities available on modern supercomputer systems. As a consequence, the analysis algorithms and tools based on PEARL will usually be parallel applications in their own right with expected scalability improvements compared to serial versions.

The originally envisaged usage model of PEARL assumes a one-to-one mapping between analysis and target-application processes. That is, for every process of the target application, one analysis process responsible for the trace data of this application process is created. However, the PEARL library itself imposes no restrictions on how many traces can be handled by a single analysis process, as long as sufficient system resources (especially memory) are available. It is even possible to implement sequential tools based on PEARL, processing the local traces one after another.

The trace data is exposed to the user through two fundamental classes, namely `GlobalDefs` and `LocalTrace`. Before describing both classes in more detail, the required organization of trace files and how it is established are explained.

4.2 Trace File Organization

To generate trace data suitable for PEARL, we have modified the original KOJAK measurement system. An essential change has been omitting the merge step and storing event data and the definitions of entities referenced by events in separate types of files, which correspond to the classes `GlobalDefs` and `LocalTrace` mentioned above.

Event data stored in trace files refer to static program entities, such as the code regions entered or left. However, to avoid redundancy and save storage space, event records contain only identifiers referencing these entities and the identifiers are defined separately. During measurement each process assigns local identifiers to these entities, and subsequently uses these local identifiers in event records whenever the corresponding entities are referenced.

Immediately after program execution, the measurement system unifies the local definitions and generates a single global definitions file, where each entity is assigned a global identifier. In addition, to allow the conversion of local into global identifiers, the measurement system creates one mapping table per process. In this way, the actual event files, which still contain local identifiers, need not be rewritten and costly I/O can be avoided. This unification step was previously performed using a separate executable, but has recently been fully integrated it into the measurement system itself. A more comprehensive description of these mechanisms can be found in [13].

4.3 Accessing Global Definitions

In the context of the PEARL library interface, information about static program entities that can be shared between all processes or threads of a parallel trace analysis tool is represented by the class `GlobalDefs`. That is, every process has to create a single instance of this class for each experiment it analyzes, which on instantiation reads the corresponding global definitions file generated by the measurement system. All processes read the same file, because they share the same set of global definitions.

The `GlobalDefs` instance provides the user with details regarding the hierarchical structure of the computer system used during trace file generation, consisting of machine, node, process, and thread descriptions as well as their relationships, such as the topological distribution (either from a logical point of view or with respect to the hardware). In addition, it offers ways to query information on groups of locations in the system hierarchy (i.e., threads or processes), which are, for instance, used to specify MPI communicators.

Moreover, the `GlobalDefs` object stores the details of instrumented code regions and call sites, as well as the global call tree of the application that is currently analyzed. This global call tree will be generated by the measurement system at the end of execution of the target application and stored in the global definitions file. Alternatively, the PEARL library provides functionality to reconstruct the global call tree during the trace analysis as an optional preprocessing step. However, this reconstruction can not be performed before the entire event-trace data has been loaded into memory. Note that different from EARL, the call tree is no longer defined in terms of links between individual events (i.e., pointer attributes), but in a separate data structure which simplifies the handling of call-path information.

4.4 Accessing Event Data

In addition to the `GlobalDefs` object, each analysis process (or thread in case of multithreaded applications) has access to one local event trace represented by an instance of the class `LocalTrace`. Since we assume that the internal in-memory representation of a local trace is smaller than the memory available to a single process on a parallel machine, the entire local trace can be kept in main memory, relaxing the aforementioned limitations resulting from strict forward analysis. In other words, the PEARL library can provide performance-transparent access to individual events plus local execution state information.

To make sure that our assumption of being able to keep the entire trace data in memory is not too restrictive for the future, the `LocalTrace` interface provided by PEARL is designed in such a way that it allows to select between different underlying trace data structures, which can be chosen when the library is compiled. At present, PEARL offers (i) a linear list with full functionality and (ii) rudimental support for cCCG graphs. The latter option will be extended and further investigated. To support very long traces, it would even be possible to add an EARL-like backend using a sliding-window approach and sophisticated buffering mechanisms.

While reading the event trace into memory, the `LocalTrace` object automatically performs two important operations: First, it corrects the timestamps of the individual events using linear interpolation to – at least partially – compensate for unsynchronized clocks. And second, it "globalizes" all identifiers used in the trace file by creating references to the corresponding static program entities provided by the single `GlobalDefs` instance, using the per-process mapping tables mentioned in Section 4.2. That is, the event objects created from the event records provide pointers into the same set of objects. After these on-the-fly transformations, which are completely transparent to the user, the instances of class `LocalTrace` provide a unified view of the event data with respect to timestamps and references to global definition objects. This is especially useful when exchanging event data between processes, as described in Section 4.5.

Individual events of local traces can be accessed through the `Event` class which provides access to all possible event attributes, following the *Composite* design pattern [14]. For navigating through the local trace, this class also exposes *Iterator* semantics available via simple `operator--()` and `operator++()` methods. With respect to the iterator functionality, the two classes `LocalTrace` and `Event` provide an interface that is very similar to that of the C++ Standard Template Library (STL) container classes and their corresponding iterators. For example, the `LocalTrace` class provides `begin()` and `end()` methods returning reasonable `Event` instances. In fact, it is even possible to apply STL algorithms, such as `for_each()` or `count_if()`, to local traces. Not to impose too many restrictions on the underlying data structure, we have refrained from providing event access by index. However, our experience suggests that iterator functionality in combination with traversal of pointer attributes is sufficient to implement complex applications such as a parallel trace analyzer.

In addition to the iterator functionality, instances of the `Event` class also provide pointer attributes for more sophisticated navigation tasks. As a result of the parallel in-memory event storage, pointer attributes can now also point forward, but no longer to remote events. Currently, there are pointer attributes to identify the enter and exit events of the enclosing region instance. These can be used to determine the duration of the communication operation (i.e., region instance) belonging to a given communication event. The return values of these pointer attribute methods are always new `Event` (i.e., iterator) objects that can be subject to further navigation operations. Local call stacks are easily calculated on the fly by traversing the chain of pointer attributes. Another special attribute identifies the call path of an event by providing a pointer into the global call tree. In this way, PEARL applications can easily identify events with equivalent call paths, a feature used to automatically associate bottlenecks with the call paths causing them. However, in contrast to the serial version, all other global states and pointers now have to be established on the application level using the event exchange operations discussed below.

4.5 Exchanging Event Data between Processes

To facilitate inter-process analysis of communication patterns, PEARL provides means to conveniently exchange one or more events between processes. Remote events received from other processes are represented by a class `RemoteEvent`, which provides a public interface very similar to the class `Event`, but without iterator semantics and pointer attributes, since we do not have full access to the remote event trace.

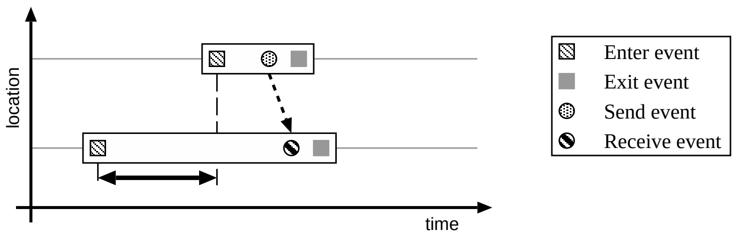
There are generally two modes of exchanging events: point-to-point and collective. Point-to-point exchange allows a `RemoteEvent` instance to be created with arguments specifying the source process, a communicator, and a message tag. In addition, the corresponding source process has to invoke a send method on the local `Event` object to be transferred.

Moreover, the exchange of multiple events can be accomplished in one batch by first collecting local events in an object of the class `EventSet` on the sender's side and instantiating an object of class `RemoteEventSet` on the receiver's side by supplying message parameters to the constructor. Each event stored in these sets is identified by a numeric identifier which can be used to assign a role to it, for example, to distinguish a particular constituent of a pattern. However, both set classes are able to transparently handle multiple role identifiers for one and the same event to avoid sending its data twice.

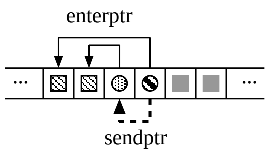
Unlike point-to-point communication, the collective event exchange provided by PEARL has the form of a reduction operation that identifies the earliest or latest event (i.e., minimum or maximum operation based on the timestamps) in participating processes' local `EventSets`, and creates a corresponding instance of class `RemoteEvent` for those processes.

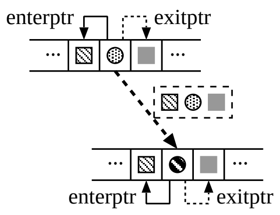
5 Example: Scalable Parallel Trace Analysis

The strength of our sequential interface EARL has been the provision of truly parallel abstractions that allows access to higher-level structures, such as messages and collective operations, which requires the ability to match corresponding events across several processes. In the case of our new parallel interface, this is more difficult, since matching those events incurs costly communication. To minimize this communication overhead, the intended usage of PEARL is that of a *replay-based* analysis. This approach has been successfully utilized to implement a parallel trace-based performance analyzer functionally almost equivalent to the MPI-1 part of the aforementioned serial trace analyzer EXPERT.



(a) Illustration of the *Late Sender* pattern





(b) Sequential trace (c) Distributed trace

Figure 1. Comparison of old and new approaches: (a) Illustration of the *Late Sender* pattern. (b) In a sequential EARL-based analyzer, the detection of the *Late Sender* pattern is triggered by a receive event. All other relevant events are found through pointer attributes. (c) In the parallel, i.e., PEARL-based approach, an `EventSet` is created whenever a send event is found by an analysis process. This send event and the associated enter and exit events are added to the set. This set is then sent to the corresponding receiver, which in turn instantiates a `RemoteEventSet` when the corresponding receive event is reached. Now the receiving process has access to all relevant constituents and can therefore verify the existence of the *Late Sender* pattern.

The central idea behind a replay-based analysis is to analyze each communication operation using an operation of similar type, that is, by reconstructing the original communication behavior of the target application currently under investigation. For example, to analyze a message transfer in point-to-point mode, the related event data is also exchanged using a single point-to-point operation.

To accomplish the analysis, the new analyzer is executed on as many CPUs as used by the target application – assuming a one-to-one mapping between analysis and application processes. That is, for every process of the target application, one analysis process responsible for the trace data of this application process is created. Using a one-to-one mapping, the analysis can be efficiently carried out immediately after trace generation as part of the same job. In the future, however, we plan to relax this model and allow a smaller number of analysis processes which might be useful if the analysis should be performed on a different system.

As a first step, each process of the analyzer instantiates a `GlobalDefs` as well as a `LocalTrace` object, thereby loading the corresponding trace data into main memory. Next, they traverse their local traces in parallel using the iterator functionality provided by the `LocalTrace` and `Event` classes and meet at the synchronization points of the target application by replaying the original communication. For this purpose we use the event data exchange abstractions described in Section 4.5. See Figure 1 for an exemplary illustration of how this principle works for the *Late Sender* pattern.

Since the PEARL library provides performance-transparent access to all events of a local trace, the analysis is no longer restricted to a pure forward analysis. That is, PEARL offers the possibility to not only exchange the data of a communication event (and potentially also the enter event of the surrounding function call accessible via a pointer attribute), but also the data of the corresponding exit event or any other event occurring in the "future". Our current prototype implementation of the parallel analyzer does not yet take advantage of this fact, but this is likely to change in subsequent versions.

Using the replay-based analysis approach implemented with PEARL, we were able to analyze execution traces of a parallel-tree application called PEPC-B running on 1,024 CPUs and the ASCI benchmark SMG2000 running on up to 16,384 CPUs. Thereby, the largest data set consisted of more than 40 billion events, which amounted to approximately 230 GBytes of disk space. By contrast, sequentially analyzing such a huge amount of data using an EARL-based analyzer was impractical. A more elaborate discussion of the parallel analyzer and the experimental results can be found in [15].

6 Conclusion and Future Work

This paper presented the design of a new parallel trace data access library called PEARL. Instead of using a single and potentially large global trace file, as was the case for our previous serial approach, the new library operates on multiple process-local trace files. This allows effective exploitation of the distributed memory and processing capabilities of modern supercomputing systems for parallel trace-analysis algorithms and tools.

The library offers basic functionality to easily access event-trace data, following well-known design principles. Because of the distributed data storage scheme, the entire event trace is held in main memory, thus yielding performance-transparent access to individual events. In addition, the interface provides a global view of static program entities referenced by the events, such as code regions or communicators, and of the call tree. Compared to its serial predecessor EARL, PEARL's storage scheme and usage model allows pointer attributes to point forward in time, which gives tool builders more flexibility in recognizing event patterns. On the other hand, PEARL no longer offers global abstractions, such as pointer attributes pointing to other processes or execution state. These have been replaced by mechanisms to conveniently exchange events between processes that, when used in conjunction with the concept of parallel reply, can provide almost equivalent but significantly more scalable trace-analysis functionality. Moreover, the basic design of the library offers the option of selecting between different trace data structures when the library is compiled. In this context, we are planning to fully implement support for the tree-based cCCG data structure and to explore its use for automatic performance analysis.

As an example of a parallel trace analysis application based on the PEARL library, we have outlined some details of our current prototype implementation of a scalable performance analyzer. The analyzer and the underlying PEARL library at present focus only on MPI-1 applications, however, we intend to extend them to support other parallel programming paradigms, such as MPI-2 and OpenMP. Finally, we plan to exploit the advantages of efficient event access to implement more sophisticated patterns that are impractical to recognize within the serial EARL framework.

References

1. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer **12** (1996) 69–80
2. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: DiP: A parallel program development environment. In: Euro-Par'96 Parallel Processing. Volume 1124 of LNCS, Springer (1996) 665–674
3. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture **49** (2003) 421–439
4. Wolf, F., Mohr, B., Dongarra, J., Moore, S.: Efficient pattern search in large traces through successive refinement. In: Euro-Par 2004 Parallel Processing. Volume 3149 of LNCS, Springer (2004) 47–54
5. Wolf, F., Mohr, B.: EARL - A programmable and extensible toolkit for analyzing event traces of message passing programs. In: High-Performance Computing and Networking. Volume 1593 of LNCS, Springer (1999) 503–512
6. Wolf, F., Freitag, F., Mohr, B., Moore, S., Wylie, B.J.N.: Large event traces in parallel performance analysis. In: Proc. 8th Workshop on Parallel Systems and Algorithms. LNI, Gesellschaft für Informatik (2006) 264–273
7. Wu, C.E., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E., Gropp, W.: From trace generation to visualization: A performance framework for distributed parallel systems. In: SC2000, IEEE Computer Society (2000)
8. Freitag, F., Caubet, J., Labarta, J.: On the scalability of tracing mechanisms. In: Euro-Par 2002 Parallel Processing. Volume 2400 of LNCS, Springer (2002) 97–104
9. Brunst, H., Nagel, W.E.: Scalable performance analysis of parallel systems: Concepts and experiences. In: Parallel Computing: Software Technology, Algorithms, Architectures and Applications. Volume 13, Elsevier (2004) 737–744
10. Miller, B.P., Clark, M., Hollingsworth, J.K., Kierstead, S., Lim, S.S., Torzewski, T.: IPS-2: The second generation of a parallel program measurement system. IEEE Transactions on Parallel and Distributed Systems **1** (1990) 206–217
11. Knüpfer, A., Nagel, W.E.: Construction and compression of complete call graphs for post-mortem program trace analysis. In: Proc. Int'l Conf. on Parallel Processing, IEEE Computer Society (2005) 165–172
12. Wolf, F.: EARL – API documentation. Technical Report ICL-UT-04-03, University of Tennessee, Innovative Computing Laboratory (2004)
13. Wylie, B.J.N., Wolf, F., Mohr, B., Geimer, M.: Integrated runtime measurement summarisation and selective event tracing for scalable parallel execution performance diagnosis. In: Proc. Workshop on State-of-the-Art in Scientific and Parallel Computing. (2006)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1995)
15. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Volume 4192 of LNCS, Springer (2006) 303–312