The SCALASCA performance toolset architecture

Markus Geimer[1], Felix Wolf[1,2], Brian J.N. Wylie[1],
Erika Ábrahám[1], Daniel Becker[1,2], Bernd Mohr[1]

[1] Forschungszentrum Jülich
Jülich Supercomputing Centre
52425 Jülich, Germany

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

{m.geimer, f.wolf, b.wylie}@fz-juelich.de
{e.abraham, d.becker, b.mohr}@fz-juelich.de
www.scalasca.org

Abstract

SCALASCA is a performance toolset that has been specifically designed to analyze parallel application execution behavior on large-scale systems. It offers an incremental performance-analysis procedure that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. Distinctive features are its ability to identify wait states in applications with very large numbers of processes and combine these with efficiently summarized local measurements. In this article, we review the current toolset architecture, emphasizing its scalable design and the role of the different components in transforming raw measurement data into knowledge of application execution behavior. The scalability and effectiveness of SCALASCA are then surveyed from experience measuring and analyzing real-world applications on a range of computer systems.

1 Introduction

World-wide efforts of building parallel machines with performance levels in the petaflops range acknowledge that the requirements of many key applications can only be met by the most advanced custom-designed large-scale computer systems. However, as a prerequisite for their productive use, the HPC community needs powerful and robust performance-analysis tools that make the optimization of parallel applications both more effective and more efficient. Such tools can not only help improve the scalability characteristics of scientific codes and thus expand their potential, but also allow domain experts to concentrate on the underlying science rather than to spend a major fraction of their time tuning their application for a particular machine.

As the current trend in microprocessor development continues, this need will become even stronger in the future. Facing increasing power dissipation and with little instruction-level parallelism left to exploit, computer architects are realizing further performance gains by using larger numbers of

moderately fast processor cores rather than by increasing the speed of uni-processors. As a consequence, supercomputer applications are required to harness much higher degrees of parallelism in order to satisfy their growing demand for computing power. With an exponentially rising number of cores, the often substantial gap between peak performance and that actually sustained by production codes [10] is expected to widen even further. Finally, increased concurrency levels place higher scalability demands not only on applications but also on parallel programming tools [15]. When applied to larger numbers of processes, familiar tools often cease to work satisfactorily (e.g., due to escalating memory requirements, limited I/O bandwidth, or renditions that fail).

Developed at Jülich Supercomputing Centre in cooperation with the University of Tennessee as the successor of KOJAK [16], SCALASCA is an open-source performance-analysis toolset that has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but is also well-suited for small- and medium-scale HPC platforms. SCALASCA supports an incremental performance-analysis procedure that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. A distinctive feature is the ability to identify wait states that occur, for example, as a result of unevenly distributed workloads. Especially when trying to scale communication-intensive applications to large processor counts, such wait states can present severe challenges to achieving good performance. Compared to KOJAK, SCALASCA can detect such wait states even in very large configurations of processes using a novel parallel trace-analysis scheme [5].

In this article, we review the current SCALASCA toolset architecture, emphasizing its scalable design. After covering related work in Section 2, we give an overview of the different functional components in Section 3 and describe the interfaces between them. In Sections 4, 5, and 6, we highlight individual aspects, such as application instrumentation, measurement and analysis of performance data, and presentation of analysis results. We survey the role of SCALASCA in the analysis of real-world applications on a range of leadership (and smaller) HPC computer systems in Section 7, before presenting ongoing and future activities in Section 8.

2 Related Work

Developers of parallel applications can choose from a variety of performance-analysis tools, often with overlapping functionality but still following distinctive approaches and pursuing different strategies on how to address today's demand for scalable performance solutions. From the user's perspective, the tool landscape can be broadly categorized into monitoring tools which present performance analysis during measurement versus those providing postmortem analysis. On a technical level, one can additionally distinguish between direct instrumentation and interrupt-based event measurement techniques.

Based on postmortem analysis presentation of direct measurements, that are traced or summarized at runtime, SCALASCA is closely related to TAU [12]. Both tools can interoperate in several modes: calls to the TAU measurement API can be redirected to the SCALASCA measurement system, making TAU's rich instrumentation capabilities available to SCALASCA users. Likewise, SCALASCA's summary and trace-analysis reporting can leverage TAU's profile visualizer and take advantage of the associated performance database framework. Compared to TAU's architecture, SCALASCA

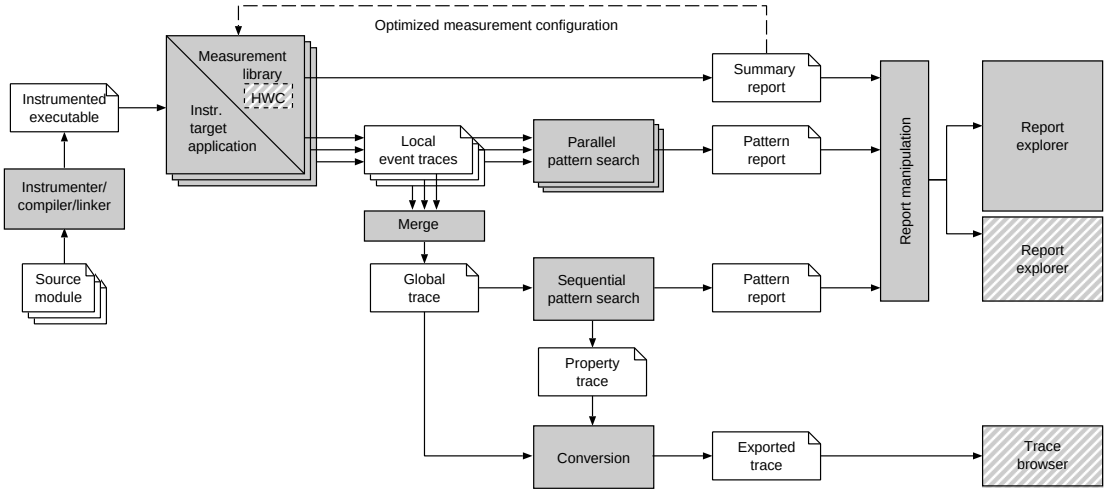


Figure 1: Schematic overview of the performance data flow in SCALASCA. Grey rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs, files or data objects running or being processed in parallel. Hatched boxes represent optional third-party components.

advocates a closer integration of summarization and tracing capabilities in a single instrumented executable and measurement experiment, and unifying local definition identifiers used in both summaries and traces in a uniform manner.

Furthermore, trace file converters connect SCALASCA to trace browsers, such as Paraver [7] and Vampir [9]. Like SCALASCA, the VampirServer architecture improves scalability through parallel trace access mechanisms, albeit targeting a 'serial' human client in front of a graphical trace browser rather than fully automatic and parallel trace analysis as provided by SCALASCA. Paraver, in contrast, favors trace-size reduction using a system of filters that eliminates dispensable features and summarizes unnecessary details using a mechanism called soft counters.

On the other hand, HPCToolkit [8] is an example of a postmortem analysis tool that generates statistical profiles from interval timer and hardware-counter overflow interrupts. Its architecture integrates analyses of both the application binary and the source code to allow a more conclusive evaluation of the profile data collected. Monitoring tools, such as Paradyn [11] and Periscope [2] perform dynamic instrumentation and evaluate performance data while the application is still running. To ensure scalable communication between tool backends and frontend, their architectures employ hierarchical networks that facilitate efficient reduction and broadcast operations.

3 Overview

The current version of SCALASCA supports measurement and analysis of the MPI, OpenMP and hybrid programming constructs most widely used in highly-scalable HPC applications written in C/C++ and Fortran on a wide range of current HPC platforms [17]. Fig. 1 shows the basic analysis workflow supported by SCALASCA. Before any performance data can be collected, the target

application must be instrumented. When running the instrumented code on the parallel machine, the user can choose to generate a summary report ('profile') with aggregate performance metrics for individual function call-paths, and/or event traces recording individual runtime events from which a profile or time-line visualization can later be produced. Summarization is particularly useful to obtain an overview of the performance behavior and for local metrics such as those derived from hardware counters. Since traces tend to rapidly become very large, scoring of a summary report is usually recommended, as this allows instrumentation and measurement to be optimized. When tracing is enabled, each process generates a trace file containing records for its process-local events. After program termination, SCALASCA loads the trace files into main memory and analyzes them in parallel using as many CPUs as have been used for the target application itself. During the analysis, SCALASCA searches for characteristic patterns indicating wait states and related performance properties, classifies detected instances by category and quantifies their significance. The result is a pattern-analysis report similar in structure to the summary report but enriched with higher-level communication and synchronization inefficiency metrics.

Both summary and pattern reports contain performance metrics for every function call-path and process/thread which can be interactively examined in the provided analysis report explorer or with third-party profile browsers such as TAU's ParaProf. In addition to the scalable trace-analysis, it is still possible to run the sequential KOJAK analysis after merging the local trace files. The sequential analysis offers features that are not yet available in the parallel version, including extended MPI and OpenMP analyses, and the ability to generate traces of pattern instances, i.e., so-called performance-property traces. As an alternative to the automatic pattern search, the merged traces can be converted and investigated using third-party trace browsers such as Paraver or Vampir, taking advantage of their time-line visualizations and rich statistical functionality.

Fig. 2 shows a layered model of the SCALASCA architecture, highlighting the interfaces between the different parts of the system. The vertical axis represents the progress of the performance analysis procedure, starting at the top with the insertion of measurement probes into the application and ending at the bottom with the presentation of results (description on the left). On the right, the procedure is split into phases that occur before, during or after execution. The horizontal axis is used to distinguish among several alternatives on each stage. Colors divide the system into groups of components with related functionality, with hatched areas/boxes representing the user providing source-code annotations and components provided by third parties.

4 Instrumentation and Measurement

4.1 Preparation of Application Executables

Preparation of a target application executable for measurement and analysis requires that it must be *instrumented* to notify the measurement library of performance-relevant execution events whenever they occur. On most systems, this can be done completely automatically using compiler support; for all systems manual and automatic instrumentation mechanisms are offered. Instrumentation configuration and processing of source files are achieved by prefixing selected compilation commands and the final link command with the SCALASCA instrumenter, without requiring other changes to optimization levels or the build process.

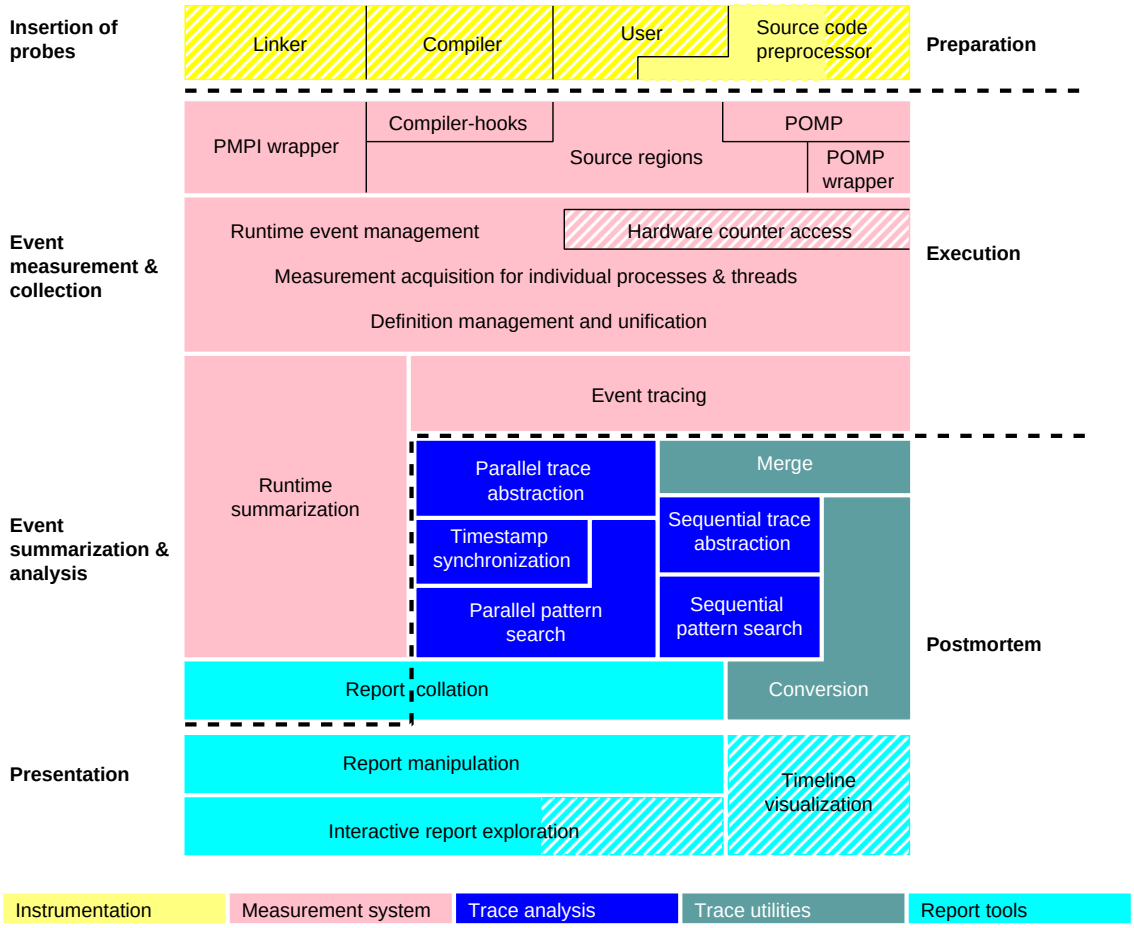


Figure 2: Layered model of the SCALASCA architecture.

Simply linking the application with the measurement library ensures that events related to MPI operations are properly captured via the standard PMPI profiling interface. For OpenMP, a source preprocessor is used which automatically instruments directives and pragmas for parallel regions, etc., based on a POMP profiling interface developed for this purpose, and most compilers are capable of adding instrumentation hooks to every function or routine entry and exit. Finally, programmers can manually add their own custom instrumentation annotations into the source code for important regions (such as phases or loops, or functions when this is not done automatically): these annotations are pragmas or macros which are ignored when instrumentation is not configured.

4.2 Measurement and Analysis Configuration

The SCALASCA measurement system [22] that gets linked with instrumented application executables can be configured via environment variables or configuration files to specify that runtime summaries and/or event traces should be collected, along with optional hardware counter metrics. During measurement initialization, a unique experiment archive directory is created to contain all

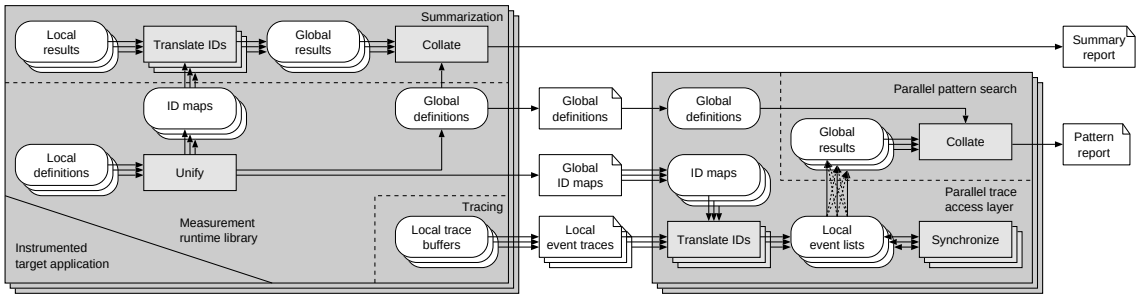


Figure 3: Detailed flow of definitions and performance data for runtime summarization (top), trace collection and parallel pattern search in traces (bottom): unification of definition identifiers and generation of identifier maps is common to both. White boxes with rounded corners denote data objects residing in memory.

of the measurement and analysis artifacts, including configuration information, log files and analysis reports. When event traces are collected, they are also stored in the experiment archive to avoid accidental corruption by simultaneous or subsequent measurements.

Measurements are collected and analyzed under the control of a nexus that determines how the application should be run, and then configures measurement and analysis accordingly. When tracing is requested, it automatically configures and executes the parallel trace analyzer with the same number of processes as used for measurement. This allows SCALASCA analysis to be specified as a command prefixed to the application execution command-line, whether part of a batch script or interactive run invocation.

In view of the I/O bandwidth and storage demands of tracing on large-scale systems, and specifically the perturbation caused by processes flushing their trace data to disk in an unsynchronized way while the application is still running, it is generally desirable to limit the amount of trace data per application process to not exceed the size of the available memory. This can be achieved via selective tracing, for example, by recording events only for code regions of particular interest or by limiting the number of timesteps during which measurements take place.

Instrumented functions which are executed frequently, while only performing a small amount of work each time they are called, have an undesirable impact on measurement. The overhead of measurement for such functions is large compared to the execution time of the (uninstrumented) function, resulting in measurement dilation, while recording such events requires significant space and analysis takes longer with relatively little improvement in quality. This is especially important for event traces whose size is proportional to the total number of events recorded. For this reason, SCALASCA offers a filter mechanism to exclude certain functions from measurement. Before starting trace collection, the instrumentation should generally be optimized based on a visit-count analysis obtained from an earlier summarization run.

4.3 Definition Unification and Analysis Collation

Measured event data refer to objects such as source code regions, call-paths, or communicators. Motivated by the desire to minimize storage requirements and avoid redundancy in traces, events

reference these objects using identifiers, while the objects themselves are defined separately. To avoid extra communication between application processes during measurement acquisition, each process may use a different local identifier to denote the same object. However, to establish a global view of the program behavior during analysis, a global set of unique object definitions must be created and local identifiers replaced with global identifiers that are consistent across all processes. This procedure is called *unification* and shown in Fig. 3.

Separate collection buffers on each process are used for definition and event records, avoiding the need to extract the definitions from a combined trace later. At measurement finalization, each rank in turn sends its definition buffers to rank zero for unification into a set of global definitions and an associated identifier mapping.

The identifier mappings are returned to each process, so that they can globalize their local analysis results during the collation of a complete summary report. First rank zero (which has the unified global definitions) prepares the report header, then it gathers the aggregated metrics for each call-path from each process and appends these incrementally, before closing the summary report [3].

When tracing is performed the global definitions and mappings are written to files, along with the dumped contents of each trace buffer. These files are subsequently read by the postmortem trace analyzer to be able to translate local object identifiers in the trace files to global identifiers used during analysis. After trace analysis is complete, collation of the analysis results and writing the pattern report is performed in the same way as for the summary report.

Although unification is a predominantly sequential operation, the distributed design takes advantage of message communication to facilitate the exchange of object definitions and the generation of mapping information while reducing expensive file I/O that would be otherwise prohibitive.

For cases where it is desired to do serial trace analysis (e.g., using the KOJAK sequential trace analyzer) or convert into another trace format (e.g., for investigation in a time-line visualization), a global trace file can be produced. The distributed trace files for each rank can be merged (using the global definitions and mappings), adding a unique location identifier to each event record when writing records in chronological order. While this can be practical for relatively small traces, the additional storage space and conversion time if often prohibitive unless very targeted instrumentation is configured or the problem size is reduced (e.g., to only a few timesteps or iterations).

5 Event Summarization and Analysis

The basic principle underlying SCALASCA performance analysis capabilities is the summarization of events, that is, the transformation of an event stream into a compact representation of execution behavior, aggregating metric values associated with individual events from the entire execution. SCALASCA offers two general options of analyzing events streams: (i) immediate runtime summarization and (ii) postmortem analysis of event traces. The strength of runtime summarization is that it avoids having to store the events in trace buffers and files. However, postmortem analysis of event traces allows the comparison of timestamps across multiple processes to identify various types of wait states that would remain undetectable otherwise. Fig. 3 contrasts both summarization techniques with respect to the flow of performance data through the system. A detailed discussion is given below, paying attention to scalability challenges and how they have been addressed.

5.1 Runtime Summarization

Many execution performance metrics can be most efficiently calculated by accumulating statistics during measurement, avoiding the cost of storing them with events for later analysis. For example, elapsed times and hardware counter metrics for source regions (e.g., routines or loops) can be immediately determined and the differences accumulated. Whereas trace storage requirements increase in proportion to the number of events (dependent on the measurement duration), summarized statistics for a call-path profile per thread have a fixed storage requirement (dependent on the number of threads and executed call-paths). SCALASCA associates metrics with unique call-paths for each thread, and updates these metrics (typically via accumulation) during the course of measurement.

Call-paths are defined as lists of visited regions (starting from an initial root), and a new call-path can be specified as an extension of a previously defined call-path to the new terminal region. In addition to the terminal region identifier and parent call-path identifier, each call-path object also has identifiers for its next sibling call-path and its first child call-path. When a region is entered from the current call-path, any child call-path and its siblings are checked to determine whether they match the new call-path, and if not a new call-path is created and appropriately linked (to both parent and last sibling). Exiting a region is then straightforward as the new call-path is the current call-path's parent call-path.

Constructing call-paths in this segmented manner provides a convenient means for uniquely identifying a call-path as it is encountered (and creating it when first encountered), and tracking changes during execution. Call-paths can be truncated at a configurable depth (to ignore deep detail, e.g., for recursive functions), and will be clipped when it is not possible to store new call-paths. When execution is complete, a full set of locally-executed call-paths are defined, and these need to be unified like all other local definitions as described previously.

A new vector of time and hardware counter metrics is acquired with every region enter or exit event. This vector of measurements is logged with the event when tracing is active, and used to determine elapsed metric values to be accumulated within the runtime summary statistics record associated with the corresponding call-path. Whereas call-path visit counts and message-passing statistics (such as numbers of synchronizations and communications, numbers of bytes sent and received) can be directly accumulated, time and hardware counter metrics require initial values (on entering a new routine) for each active frame on the call-stack to be stored so that they can be subtracted when that frame is exited (on leaving the routine). Keeping separate call-path statistics and stacks of entry metric vectors for each thread allows efficient lock-free access to the values required during measurement.

5.2 Postmortem Trace Analysis

In message-passing applications, processes often require access to data provided by remote processes, making the progress of a receiving process dependent upon the progress of a sending process. Collective synchronization is similar in that its completion requires each participating process to have reached a certain point. As a consequence, a significant fraction of the communication and synchronization time can often be attributed to wait states, for example, as a result of

an unevenly distributed workload. Especially when trying to scale communication-intensive applications to large process counts, such wait states can present severe challenges to achieving good performance. SCALASCA provides a diagnostic method that allows their localization, classification, and quantification by automatically searching event traces for characteristic patterns. A list of the patterns supported by SCALASCA including explanatory diagrams can be found on-line [6].

To accomplish the search in a scalable way, both distributed memory and parallel processing capabilities available on the target system are exploited. Instead of sequentially analyzing a single global trace file, as done by KOJAK, SCALASCA analyzes separate process-local trace files in parallel by *replaying* the original communication on as many CPUs as have been used to execute the target application itself. During the search process, pattern instances are classified and quantified according to their significance for every program phase and system resource involved. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of application processes, such pattern searches have been completed at previously intractable scales.

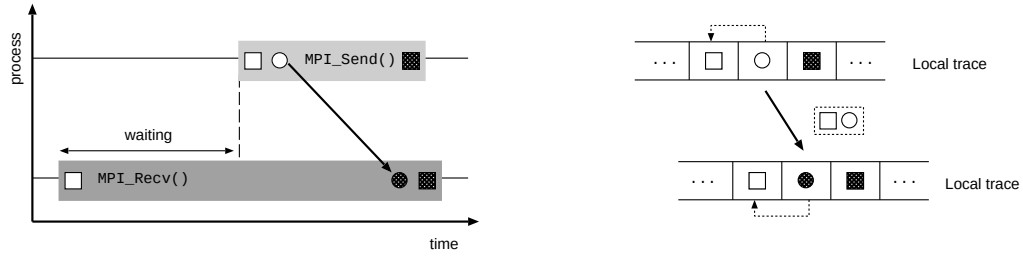
To maintain efficiency of the trace analysis process as the number of application processes increases, our architecture follows a parallel trace access model which is provided as a separate abstraction layer [4] between the parallel pattern search and the raw trace data stored on disk (Fig. 2, center). Implemented as a C++ class library, this layer offers random access to individual events as well as abstractions that help identify matching events, which is an important prerequisite for the pattern search. The main usage model of the trace-access library assumes that for every process of the target application an analysis process is created to be uniquely responsible for its trace data. Data exchange among analysis processes is then accomplished via MPI communication.

The library offers classes to represent process-local traces, events, and objects referenced by those events (e.g., regions, communicators). When instantiating a trace object, the trace data are loaded into memory while translating local into global identifiers using the mapping tables provided by the measurement system to ensure that event instances created from event records point to the correct objects (Fig. 3). Having the entire event trace kept in main memory during analysis thereby enables performance-transparent random access to individual events.

Higher-level abstractions provide the context in which an event occurs, such as the call-path or communication peers. While special event attributes store process-local context information, remote event abstractions in combination with mechanisms to exchange event data between analysis processes allow tracking of interactions across process boundaries. The actual matching of communication events is performed by exploiting MPI messaging semantics during a parallel communication replay of the event trace.

Using the infrastructure described above, the parallel analyzer traverses the local traces in parallel from beginning to end while exchanging information at synchronization points of the target application. That is, whenever an analysis process sees events related to communication or synchronization, it engages in an operation of similar type with corresponding peer processes.

As an example of inefficient point-to-point communication, consider the so-called *Late Sender* pattern (Fig. 4(a)). Here, an early receive operation is entered by one process before the corresponding send operation has been started by the other. The time lost waiting due to this situation is the time difference between the enter events of the two MPI function instances that precede the corresponding send and receive events. During the parallel replay (Fig. 4(b)), the search is triggered by the communication events on both sides. Whenever an analysis process finds a send



(a) Late Sender pattern. (b) Parallel detection.

Figure 4: Searching for the Late Sender pattern in parallel. The situation described by this pattern is shown on the left in a time-line diagram. How the different events are accessed and combined to verify its occurrence is shown on the right.

event, a message containing this event as well as the associated enter event is sent to the process representing the receiver using a non-blocking point-to-point communication. When the receiver reaches the corresponding receive event, this message is received. Together with the local receive and enter events, a *Late Sender* situation can be detected by comparing the timestamps of the two enter events and calculating the time spent waiting for the sender.

Currently, the parallel trace analysis considers only a subset of MPI operations and ignores MPI-2 RMA and OpenMP parallel operations, which can alternatively be analyzed via sequential trace analysis of a merged trace. Finally, automatic trace analysis of OpenMP applications using dynamic, nested and guarded worksharing constructs is not yet possible.

To allow accurate trace analyses on systems without globally synchronized clocks, SCALASCA provides the ability to synchronize inaccurate timestamps postmortem. Linear interpolation based on clock offset measurements during program initialization and finalization already accounts for differences in offset and drift, assuming that the drift of an individual processor is not time dependent. This step is mandatory on all systems without a global clock, such as Cray XT and most PC or compute blade clusters. However, inaccuracies and drifts varying over time can still cause violations of the logical event ordering that are harmful to the accuracy of our analysis. For this reason, SCALASCA compensates for such violations by shifting communication events in time as much as needed while trying to preserve the length of intervals between local events [1]. The logical synchronization is currently optional and should be performed if the trace analysis reports (too many) violations of the logical event order.

6 Report Generation, Manipulation, and Exploration

Summary and pattern reports are XML files, written to disk by a single process while gathering the necessary information from the remaining application or trace-analysis processes using MPI collective communication. Since the size of the report may exceed the memory capacity of the writer process, the report is created incrementally, alternating between gathering and writing smaller subsets of the overall data. Compared to an initial prototype of SCALASCA, the speed of writing reports has been substantially increased by eliminating large numbers of temporary files [3].

Reports can be combined or manipulated [13] to allow comparisons or aggregations, or to focus the analysis on specific extracts of a report. Specifically, multiple reports can be merged or averaged, the difference between two reports calculated, or a new report generated after eliminating uninteresting phases (e.g., initialization) to focus the analysis on a selected part of the execution. These utilities each generate new reports as output that can be further manipulated or viewed like the original reports that were used as input. The library for reading and writing the XML reports also facilitates the development of utilities which process them in various ways, such as the extraction of measurements for each process or their statistical aggregation in metric graphs.

To explore their contents, reports can be loaded into an interactive analysis report explorer [3]. Recently, the explorer's capacity to hold and display data sets has been raised by shrinking their memory footprint and interactive response times have been reduced by optimizing the algorithms used to calculate aggregate metrics.

7 Survey of Experience

Early experience with SCALASCA was demonstrated with the ASC benchmark SMG2000, a semi-coarsening multi-grid solver which was known to scale well on BlueGene/L (in weak scaling mode where the problem size per process is constant) but pose considerable demands on MPI performance analysis tools due to huge amounts of non-local communication. Although serial analysis with the KOJAK toolkit became impractical beyond 256 processes, due to analysis time and memory requirements, the initial SCALASCA prototype was already able to complete its analysis of a 2048-process trace in the same time [5].

Encouraged by the good scalability of the replay-based trace analysis, which was able to effectively exploit the per-process event traces without merging or re-writing them, bottlenecks in unifying definition identifiers and collating local analysis reports were subsequently addressed, and trace collection and analysis scalability with this SMG2000 benchmark extended to 16,384 processes on BlueGene/L and 22,528 processes on Cray XT3/4 [19, 21]. The latter traces amounted to 4.9TB, and 18GB/s was achieved for the final flush of the trace buffers to disk, despite the need to work around Lustre filesystem limitations on the number of files written simultaneously.

Identifier unification and map creation still took an unacceptably long time, particularly for runtime summarization measurements where large traces are avoided, however, straightforward serial optimizations have subsequently reduced this time by a factor of up to 25 (in addition to savings from creating only two global files rather than two files per process). Furthermore, the number of files written when tracing has been reduced to a single file per rank (plus the two global files), which is written only once directly into a dedicated experiment archive. Filesystem performance is expected to continue to lag behind that of computer systems in general, even when parallel I/O is employed, therefore elimination of unnecessary files provides benefits that grow with scale and are well-suited for the future.

As a specific example, SMG2000 using 65,536 processes on BlueGene/P has been analyzed with the latest (1.0) version of SCALASCA. The uninstrumented version of the application ran in 17 minutes, including 12 minutes to launch this number of processes. From an initial runtime summary of the fully-instrumented application, where execution time was dilated 25% to 6 minutes, a file

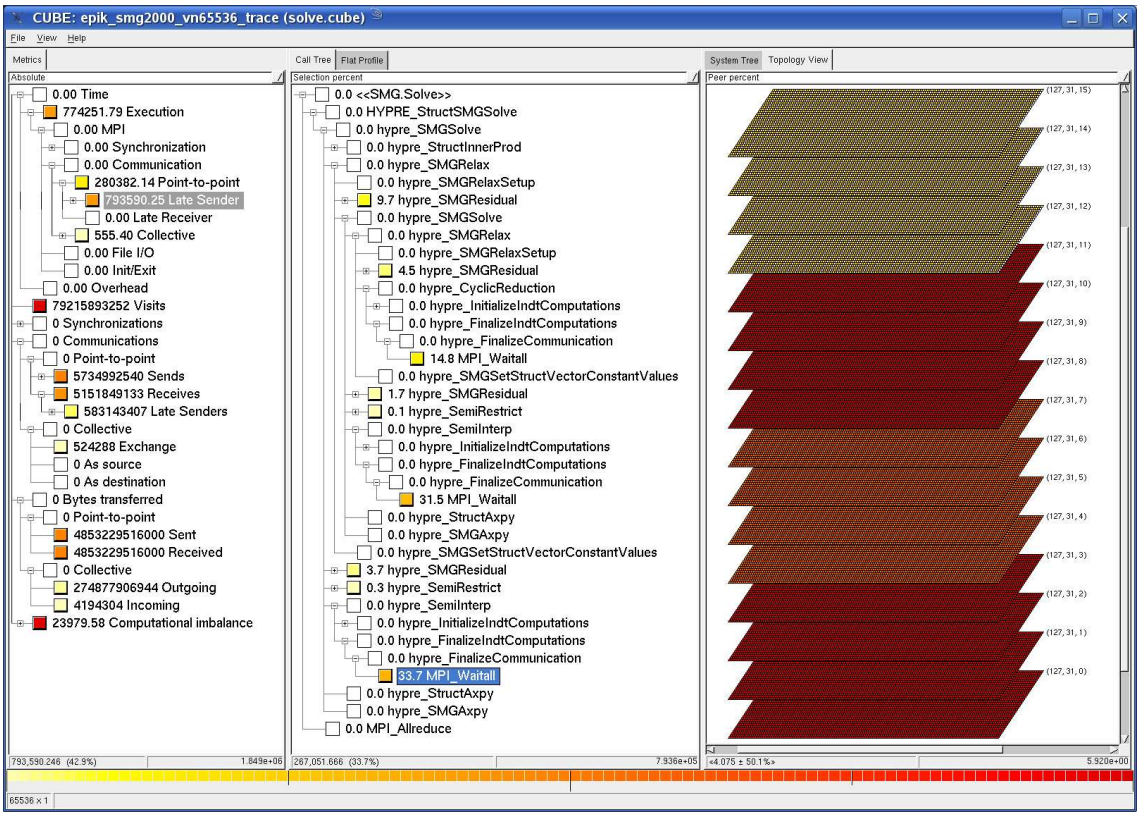


Figure 5: SCALASCA presentation of the solver extract of the trace analysis report for 64k-process measurement of SMG2000 on a 16-rack BlueGene/P, with *Late Sender* time selected from the metrics pane (left) amounting to 43% of total time (compared to 42% for *Execution* time excluding MPI). One third of *Late Sender* time is attributed to one MPI_Waitall from the call-tree pane (centre), and its distribution across the 65,536 processes shown on the BlueGene physical topology (right). Metric values are colour-coded according to the scale at the bottom to facilitate distinction of high (dark) and low (light) severities in both tree and topology displays.

listing a number of purely-computational functions to be filtered from the trace was determined. Collection and analysis of the resulting 3.33TB trace took 100 minutes (two-thirds collection and one-third analysis), including 30 minutes for the two launches. Unifying identifier definitions and writing associated maps took 12 minutes, whereas writing the traces in parallel to GPFS achieved 6.2GB/s and took 10 minutes: the major bottleneck, however, was the 25 minutes required to initially open and create one trace file per process. The solver section of the resulting 2.3GB analysis report was subsequently extracted, and Figure 5 shows the distribution of the *Late Sender* metric for the call-path taking the longest time. With its 50% standard deviation there is clearly a significant imbalance that closely corresponds to the physical racks of the BlueGene/P system. Furthermore, only 0.2% of the total number of late senders are on this call-path that requires one-third of the total *Late Sender* time, so there is potentially a great benefit from addressing this localized inefficiency, e.g., using a better mapping of processes to processors.

Beyond relatively simple benchmark kernels, SCALASCA has also successfully been used to analyze and tune a number of locally-important applications. The XNS simulation of flow inside a blood pump, based on finite-element mesh techniques, was analyzed using 4,096 processes on BlueGene/L, and after removing unnecessary synchronizations from critical scatter/gather and scan operations performance improved more than four-fold [20]. On the MareNostrum blade cluster, the WRF2 weather research and forecasting code was analyzed using 2,048 processes, and identified occasional problems with seriously imbalanced exits from MPI_Allreduce calls that significantly degraded overall application performance [18]. In both cases, the high-level call-path profile readily available from runtime summarization was key in identifying general performance issues that manifest at large scale, related to performance problems that could subsequently be isolated and understood with targeted trace collection and analysis.

The SPEC MPI2007 suite of 13 benchmark codes from a wide variety of subject areas have also been analyzed with SCALASCA with up to 1,024 processes on an IBM SP2 p690+ cluster [14]. Problems with several benchmarks that limited their scalability (sometimes to only 128 processes) were identified, and even those that apparently scaled well contained considerable quantities of waiting times indicating possible opportunities for performance and scalability improvement. Although runtime summarization could be effectively applied to fully-instrumented runs of the entire suite of benchmarks, analysis of complete traces was only possible for 12 of the set, due to the huge number of point-to-point communications done by the 121.pop2 benchmark: since execution behavior was repetitive this allowed 2000 instead of 9000 timesteps to be specified for a representative shorter execution.

While large-scale tests are valuable to demonstrate scalability, more important has been the effective use of the SCALASCA toolset by application developers' on their own codes, often during hands-on tutorials and workshops, where the scale is typically relatively small but there is a great diversity of computer systems (e.g., Altix, Solaris and Linux clusters, as well as leadership HPC resources). Feedback from users and their suggestions for improvements continue to guide SCALASCA development.

8 Outlook

Our general approach is to first observe parallel execution behavior on a coarse-grained level and then to successively refine the measurement focus as new performance knowledge becomes available. Future enhancements will aim at both further improving the functionality and scalability of the SCALASCA toolset. Completing support for OpenMP and the missing features of MPI to eliminate the need for sequential trace analysis is a primary development objective.

Using more flexible measurement control, we are striving to offer more targeted trace collection mechanisms, reducing memory and disk space requirements while retaining the value of trace-based in-depth analysis. In addition, while the current measurement and trace analysis mechanisms are already very powerful in terms of the number of application processes they support, we are working on optimized data management and workflows that will allow us to master even larger configurations. These might include truly parallel identifier unification, trace analysis without file I/O, and using parallel I/O to write analysis reports. Since parallel simulations are often

iterative in nature, and individual iterations can differ in their performance characteristics, another focus of our research is therefore to study the temporal evolution of the performance behavior as a computation progresses.

References

[1] D. Becker, R. Rabenseifner, and F. Wolf. Timestamp synchronization for event traces of large-scale message-passing applications. In *Proc. 14th European PVM and MPI Conference (EuroPVM/MPI, Paris, France)*, Lecture Notes in Computer Science 4757, pages 315–325. Springer, September 2007.

[2] K. Fürlinger and M. Gerndt. Distributed application monitoring for clustered SMP architectures. In *Proc. European Conference on Parallel Computing (Euro-Par, Klagenfurt, Austria)*, Lecture Notes in Computer Science 2790, pages 127–134. Springer, August 2003.

[3] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B. J. N. Wylie. Scalable collation and presentation of call-path profile data with CUBE. In *Parallel Computing: Architectures, Algorithms and Applications: Proc. Parallel Computing (ParCo'07, Jülich/Aachen, Germany)*, volume 15, pages 645–652. IOS Press.

[4] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. J. N. Wylie. A parallel trace-data interface for scalable performance analysis. In *Proc. Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA, Umeå, Sweden)*, Lecture Notes in Computer Science 4699, pages 398–408. Springer, June 2006.

[5] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proc. 13th European PVM and MPI Conf. (EuroPVM/MPI, Bonn, Germany)*, Lecture Notes in Computer Science 4192, pages 303–312. Springer, September 2006.

[6] Jülich Supercomputing Centre. *SCALASCA performance analysis toolset documentation*. `http://www.scalasca.org/software/documentation`.

[7] J. Labarta, J. Gimenez, E. Martinez, P. Gonzalez, H. Servat, G. Llort, and X. Aguilar. Scalability of visualization and tracing tools. In *Proc. Parallel Computing (ParCo, Málaga, Spain)*, NIC series Vol. 33, pages 869–876. Forschungszentrum Jülich, September 2005.

[8] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002.

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

[10] L. Oliker, A. Canning, J. Carter, C. Iancu, M. Lijewski, S. Kamil, J. Shalf, H. Shan, E. Strohmaier, S. Ethier, and T. Goodale. Scientific application performance on candidate petascale platforms. In *Proc. Int'l Parallel & Distributed Processing Symposium (IPDPS, Long Beach, CA, USA)*, pages 1–12, 2007.

[11] P. C. Roth and B. P. Miller. On-line automated performance diagnosis on thousands of processes. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06, New York City, NY, USA)*, pages 69–80, March 2006.

[12] S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006. SAGE Publications.

[13] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *Proc. 33rd Int'l Conference on Parallel Processing (ICPP, Montreal, Canada)*, pages 63–72. IEEE Computer Society, August 2004.

[14] Z. Szebenyi, B. J. N. Wylie, and F. Wolf. SCALASCA parallel performance analyses of SPEC MPI2007 applications. In *Performance Evaluation — Metrics, Models and Benchmarks*, Lecture Notes in Computer Science 5119. Springer, July 2008.

[15] M. L. Van De Vanter, D. E. Post, and M. E. Zosel. HPC needs a tool strategy. In *Proc. 2nd Int'l Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS, St. Louis, MO, USA)*, pages 55–59, May 2005.

[16] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10–11):421–439, 2003.

[17] F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi. Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In *Proc. 2nd HLRS Parallel Tools Workshop (Stuttgart, Germany)*. Springer, July 2008. (to appear).

[18] B. J. N. Wylie. Scalable performance analysis of large-scale parallel applications on MareNostrum. In *Science and Supercomputing in Europe Report 2007*. HPC-Europa Transnational Access, CINECA, Casalecchio di Reno (Bologna), Italy, 2008. (to appear).

[19] B. J. N. Wylie and M. Geimer. Performance measurement & analysis of large-scale scientific applications on leadership computer systems with the Scalasca toolset. In *Proc. Seminar 07341 on Code Instrumentation and Modeling for Parallel Performance Analysis (CIMPPA)*. Schloss Dagstuhl, Germany, August 2007.

[20] B. J. N. Wylie, M. Geimer, M. Nicolai, and M. Probst. Performance analysis and tuning of the XNS CFD solver on BlueGene/L. In *Proc. 14th European PVM and MPI Conference (EuroPVM/MPI, Paris, France)*, Lecture Notes in Computer Science 4757, pages 107–116. Springer, September 2007.

[21] B. J. N. Wylie, M. Geimer, and F. Wolf. Performance measurement and analysis of large-scale parallel applications on leadership computing systems. *Journal of Scientific Programming*, 2008. IOS Press (to appear).

[22] B. J. N. Wylie, F. Wolf, B. Mohr, and M. Geimer. Integrated runtime measurement summarization and selective event tracing for scalable parallel execution performance diagnosis. In *Proc. Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA, Umeå, Sweden)*, Lecture Notes in Computer Science 4699, pages 460–469. Springer, June 2006.