CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:702–719 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

The Scalasca performance toolset architecture

Markus Geimer[1], Felix Wolf[1,2,3], Brian J. N. Wylie[1], Erika Ábrahám[3], Daniel Becker[1,3], Bernd Mohr[1]

[1] *Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany*
[2] *German Research School for Simulation Sciences, 52062 Aachen, Germany*
[3] *Computer Science Department, RWTH Aachen University, 52056 Aachen, Germany*

SUMMARY

Scalasca is a performance toolset that has been specifically designed to analyze parallel application execution behavior on large-scale systems with many thousands of processors. 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 to 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.

KEY WORDS: parallel computing, performance analysis, scalability

1. INTRODUCTION

Worldwide efforts at 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 one prerequisite for their productive use, the HPC community needs powerful and robust performance-analysis tools that

*Correspondence to: Felix Wolf, German Research School for Simulation Sciences, 52062 Aachen, Germany, `f.wolf@grs-sim.de`

Contract/grant sponsor: Helmholtz Association, Grant No. VH-NG-118
Contract/grant sponsor: Helmholtz Association, Grant No. VH-VI-228
Contract/grant sponsor: German Federal Ministry of Research and Education (BMBF), Grant No. 01IS07005C

make the optimization of parallel applications both more effective and more efficient. Such tools will not only help to 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. With many-core processors growing to hundreds of cores, scalability-enhancing tools will soon be needed even for deskside systems.

In order to satisfy their growing demand for computing power, supercomputer applications are required to harness unprecedented degrees of parallelism. With an exponentially rising number of cores, the often substantial gap between peak performance and that actually sustained by production codes [16] is expected to widen even further. However, increased concurrency levels place higher scalability demands not only on applications but also on parallel programming tools needed for their development [22]. 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 by Jülich Supercomputing Centre as the successor to KOJAK [23], Scalasca is an open-source performance-analysis toolset that has been specifically designed for use on large-scale systems with many thousands of cores 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 its 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 major 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 [8].

In this article, we review the current Scalasca toolset architecture, emphasizing its scalable design for the measurement and analysis of many thousands of processes or threads. 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, including first trace collection and analysis results with 292 914 processes, 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 or online tools, on the one hand, and postmortem tools, on the other hand. While the former provide immediate feedback on the performance behavior while the application is still running and − in principle − allow the user to intervene if necessary, they usually require a more complex runtime infrastructure and are therefore harder to use in batch mode. In contrast, postmortem tools only record performance behavior at runtime and then analyze it after the target program has terminated. Although with postmortem tools it is necessary to store potentially large amounts of performance data for later transfer to the analysis stage, they are more suitable for providing a global view of the entire execution as opposed to a snapshot subject to runtime analysis. On a lower technical level, one can further distinguish between direct instrumentation and interrupt-based event measurement techniques. While interrupt-based performance measurement offers simpler control of the runtime overhead by allowing adjustments of the interrupt interval, direct instrumentation ensures that all events of interest are seen during measurement, offering advantages for the global analysis of communication and synchronization operations as well as for the characterization of shorter intervals or of longer intervals with high temporal variability.

Based on the postmortem analysis presentation of direct measurements, traced or summarized at runtime, Scalasca is closely related to TAU [18]. 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. A major difference between TAU and Scalasca lies in the way the two tools manage local objects performance measurements refer to, such as regions or call paths. Whereas TAU defines a relatively small number of object types that are unified postmortem, Scalasca distinguishes a larger variety of object types that are unified at measurement finalization. These include, for example, communicator objects that enable the identification and analysis of individual collective operation instances in event traces.

Furthermore, trace-file converters connect Scalasca to trace browsers such as Paraver [11] and Vampir [15]. 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.

HPCToolkit [12] 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. Online tools, such as Paradyn [17] and Periscope [3], evaluate performance data while the application is still running. Both tools search for previously specified performance problems or properties. A search strategy directs performance measurements, which are successively refined based on the current findings. To

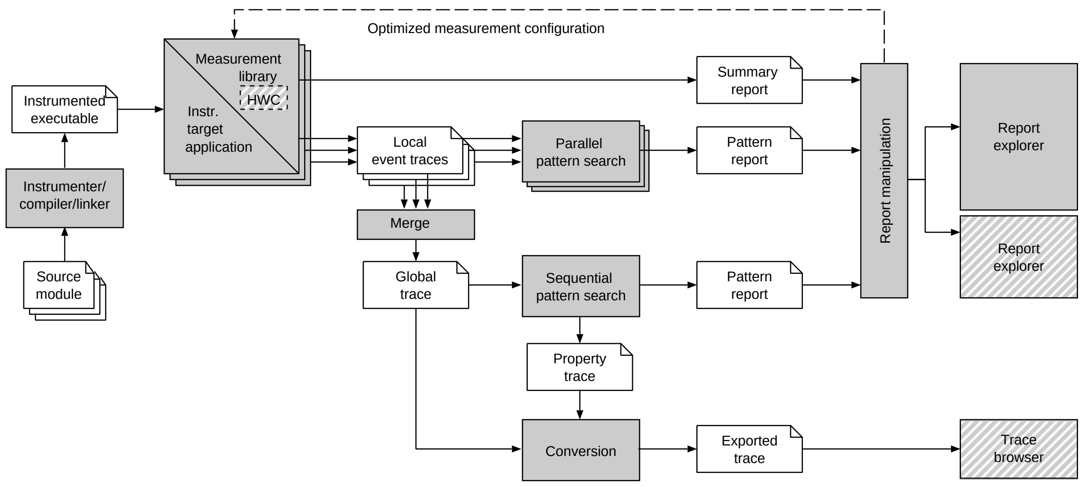


Figure 1. Schematic overview of the performance data flow in Scalasca. Gray rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs, files, or data objects running or being processed in parallel. Hatched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

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++, or Fortran on a wide range of current HPC platforms [24]. Figure 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 cores 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 and 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, such as 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. A very recent extension of Vampir now also allows the parallel Scalasca traces to be read directly without merging and then converting the local traces beforehand.

Figure 2 shows a diagram 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 target-program execution. The horizontal axis is used to distinguish among several alternatives at each stage. Groups of components with related functionality are shaded the same way, 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 all systems, a mix of manual and automatic instrumentation mechanisms is 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.

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 for parallel regions, etc., based on the POMP profiling interface [13] developed for this purpose. On most systems, the instrumentation of user functions is accomplished via compiler-supplied profiling interfaces capable of automatically adding hooks to every function or routine entry and exit. Moreover, as a portable alternative, the TAU source-code instrumentor can be configured to directly insert Scalasca measurement API calls [5]. Finally, programmers can manually add their own

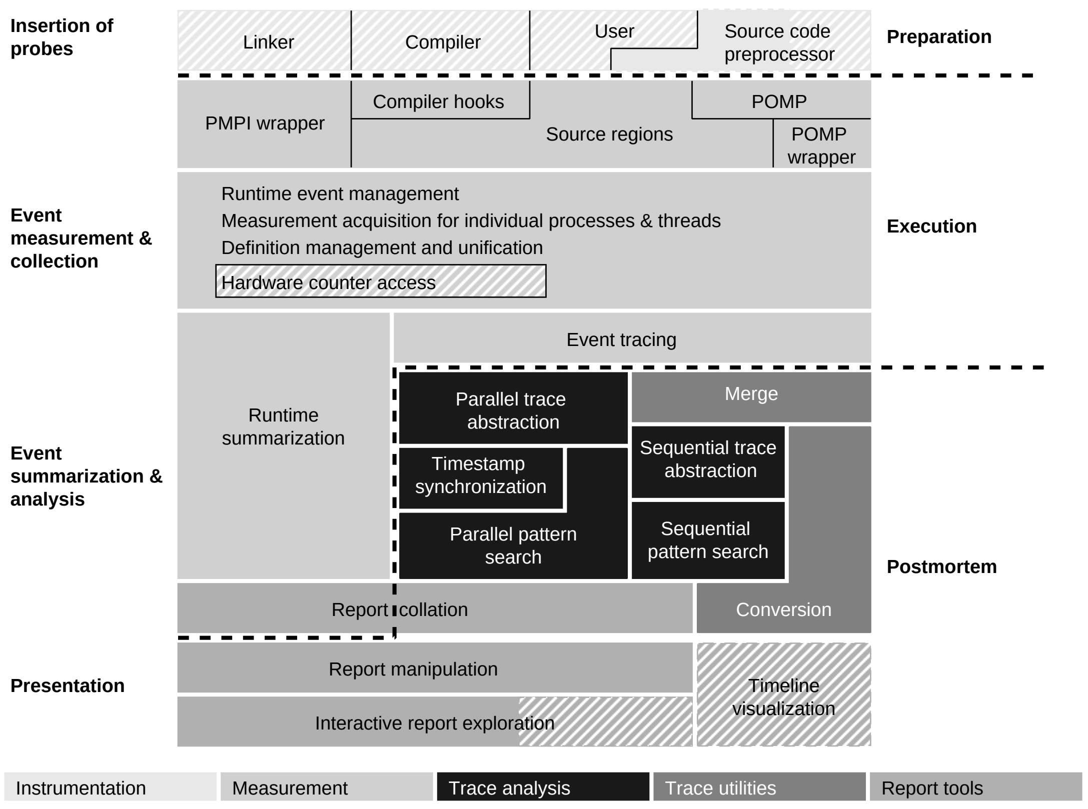


Figure 2. Scalasca architecture. Each box represents one component of the toolset, with optional third-party components shown as hatched areas. The vertical placement indicates the different phases of the analysis process, whereas alternatives in each phase are depicted next to each other horizontally. Note that the phases are categorized in two different ways, either functionally (description on the left) or temporally (separated by dashed lines, description on the right).

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 [28] linked with instrumented application executables can be configured via environment variables or configuration files to specify that runtime summaries or/and event traces should be collected, along with optional hardware counter metrics. During measurement initialization, a unique experiment archive directory is created to contain all 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 workflow manager 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 so that the size of the available memory is not exceeded. 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. The large majority of the applications we analyzed so far left enough memory for a trace buffer to record at least one iteration of the timestep loop.

Instrumented functions that 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.

Since it is roughly proportional to the frequency of measurement routine invocations, the execution time dilation induced by the instrumentation is highly application dependent. Whereas the frequency of user-code events can be reduced using the aforementioned filter mechanism, throttling the generation of communication-related events is much harder to accomplish. Moreover, if the communication frequency rises with increasing numbers of processes then the observable dilation might also become a matter of scale.

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, whereas 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 Figure 3 as part of the measurement runtime library.

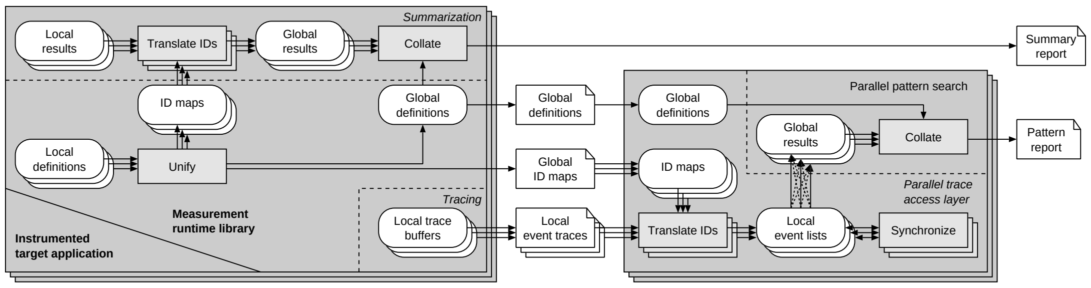


Figure 3. Detailed flow of definitions and performance data for runtime summarization and trace collection and parallel pattern search in traces: unification of definition identifiers and generation of identifier maps is common to both. Bold labels identify core components, while sub-components are separated using dashed lines and labeled in italics. White boxes with rounded corners denote data objects residing in memory.

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. Although a single buffer per process is used for definitions, each thread has a separate buffer for its event records. At measurement finalization, each rank in turn sends its definition buffer to rank zero for unification into a set of global definitions and an associated identifier mapping. Although our current unification algorithm is predominantly sequential, 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. When summarization is configured, 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. In a next step, rank zero (which has the unified global definitions) prepares the report header before it gathers the aggregated metrics for each call path from each process and appends these to the report body. 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. The MPI gather operation used for this purpose allows this procedure to take advantage of efficient tree-based algorithms employed in most MPI implementations. Compared to an initial prototype of Scalasca, the speed of writing reports has been substantially increased by eliminating large numbers of temporary files [4].

When tracing is configured, 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.

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 are 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 for 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. Figure 3 contrasts the two 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 (depending on the measurement duration), summarized statistics for a call-path profile per thread have a fixed storage requirement (depending 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.

In addition to call-path visit counts, execution times, and optional hardware counter metrics, Scalasca measurements and analyses include various MPI statistics, such as the numbers of synchronization, communication, and file I/O operations, with the associated number of bytes transferred, each broken down into collective versus point-to-point/individual, sends/writes versus receives/reads, etc. Call-path execution times separate MPI message-passing and OpenMP multithreading costs from purely local computation, and further break them down into initialization/finalization, synchronization, communication, file I/O, and thread management overheads (as appropriate). For measurements using OpenMP, additional thread idle time and limited parallelism metrics are derived assuming a dedicated core for each thread.

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.

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 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.

Synthetic OpenMP fork and join events on the master thread, which enclose the execution of each parallel region by the team members, allow worker threads to acquire the call path for the current parallel region from the master thread. Threads that do not participate in executing a parallel region are identified, and OpenMP thread team management overhead is derived. At measurement completion, the call paths executed by worker threads are locally unified with those of the master thread, before the definitions from each process are globally unified. The metrics for each thread team are then locally collated for each call path, interleaved with gathering the values from all processes in hybrid measurements, to create the summary analysis report.

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 serious challenges to achieving good performance. Scalasca provides a diagnostic method that allows the localization of wait states by automatically searching event traces for characteristic patterns. A list of the patterns supported by Scalasca including explanatory diagrams can be found on-line [10]. An example is discussed in Section 7.

To accomplish the search in a scalable way, both distributed memory and parallel processing capabilities available on the target system are exploited [8]. 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 cores 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 as the number of application processes increases, our architecture follows a parallel trace access model, which is provided as a separate abstraction layer [6] between the parallel pattern search and the raw trace data stored on disk (Figure 2, center and Figure 3, bottom right). 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. Before starting the trace analysis, 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 (Figure 3). Keeping the entire event trace in main memory during analysis thereby enables performance-transparent random access to individual events, but also limits the trace size per process and may necessitate measures such as coarsening the instrumentation or restricting the measurement to selected intervals.

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. Exploiting MPI messaging semantics, the analyzer combines matching communication or synchronization events including their contexts and checks the resulting compounds for the occurrence of wait states. Recently, we also applied this replay-based trace-analysis scheme to detect wait states occurring in MPI-2 RMA operations, using RMA communication to exchange the required information between processes [9]. In addition, we incorporated the ability to analyze traces from hybrid MPI/OpenMP applications. Although no additional OpenMP metrics are collected compared to the runtime summarization, there is the added value of the enhanced MPI analysis even for hybrid codes.

Currently, the parallel trace analysis is not yet available for pure OpenMP codes, which can alternatively be analyzed via sequential trace analysis of a merged trace. Finally, automatic trace analysis of OpenMP or hybrid MPI/OpenMP applications using dynamic, nested, and guarded worksharing constructs is not yet possible.

Traces recorded on HPC systems with globally synchronized high-resolution clocks, such as IBM Blue Gene and clusters with switch clocks, can be simply analyzed by comparing timestamps. For other systems, such as Cray XT and most PC or compute blade clusters, 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, but inaccuracies and drifts varying over time can still cause violations of the logical event order 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 to restore the logical event order while trying to preserve the length of intervals between local events [1]. This logical synchronization is currently optional and should be performed if the trace-analysis reports (too many) violations of the logical event order.

6. REPORT 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. To explore their contents, reports can be loaded into an interactive analysis report explorer (shown in the next section). Recently, the explorer's capacity to hold and display data sets has been raised by shrinking its memory footprint and interactive response times have been reduced by optimizing the algorithms used to calculate aggregate metrics [4].

Reports can be combined or manipulated to allow comparisons or aggregations, or to focus the analysis on specific extracts of a report [19]. For example, the difference between two reports can be calculated or a new report generated after eliminating uninteresting phases (e.g. initialization). 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.

7. SURVEY OF EXPERIENCE

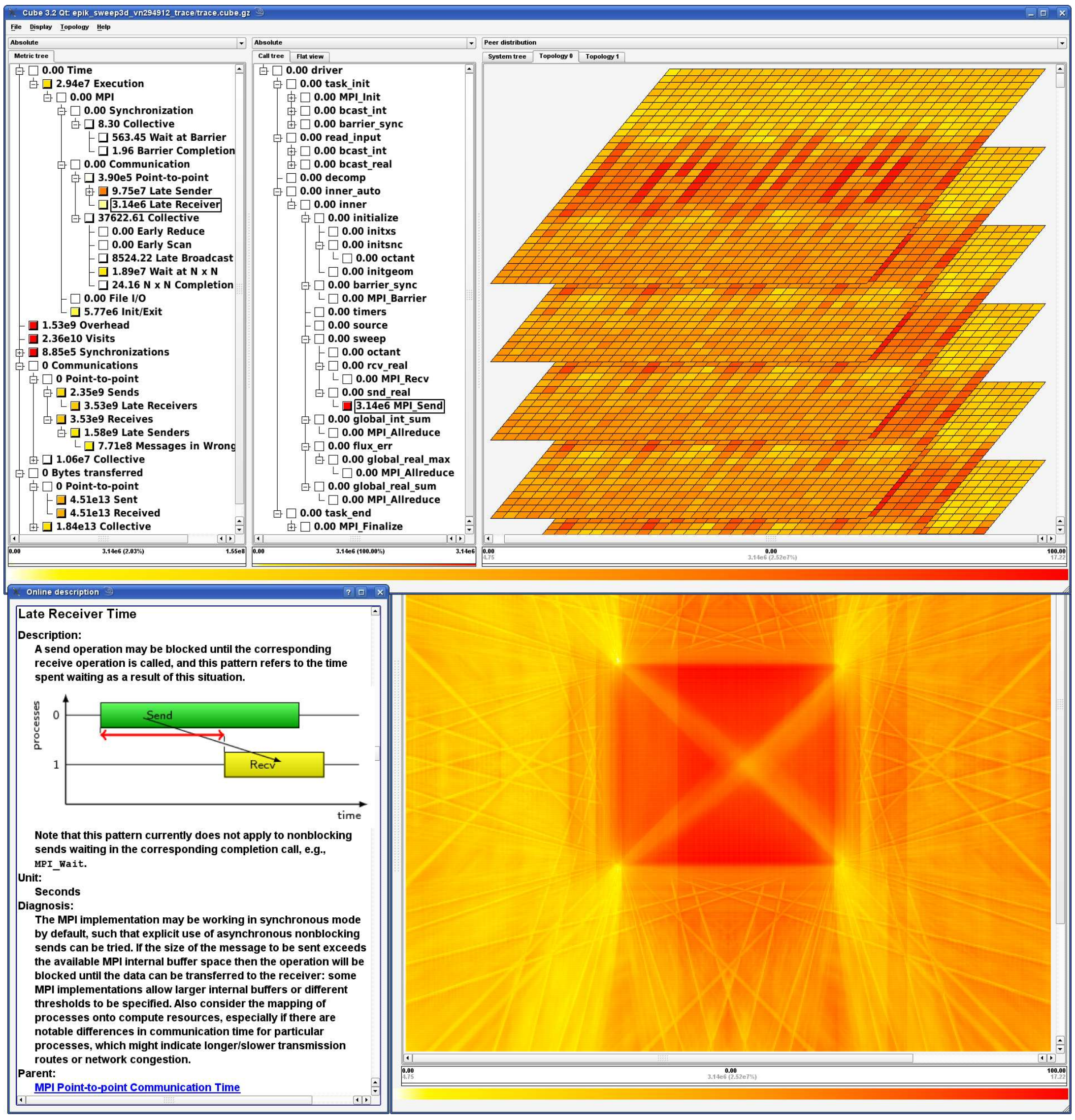
Early experience with Scalasca was demonstrated with the ASC benchmark SMG2000, a semi-coarsening multi-grid solver which was known to scale well on Blue Gene/L (in weak scaling mode where the problem size per process is constant), but which poses considerable demands on MPI performance-analysis tools 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, even the initial Scalasca prototype was able to complete its analysis of a 2048-process trace in less time than KOJAK needed for a 256-process trace [7].

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 benchmark extended to 16 384 processes on Blue Gene/L and 22 528 processes on Cray XT3/4 [27]. The latter traces amounted to 4.9TB, and 18GB/s was achieved for the final flush of the trace buffers to disk.

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. File-system performance is expected to continue to lag behind that of computer systems in general, even when parallel I/O is employed, therefore the elimination of unnecessary files provides benefits that grow with scale and are well suited for the future. This enabled Scalasca to collect and analyze SMG2000 traces with 65 536 processes on Blue Gene/P [14].

As a specific example, ASC benchmark SWEEP3D using 294 912 processes on Blue Gene/P has been analyzed with the July 2009 (1.2) release of Scalasca. The uninstrumented version of the application took 10 min to run. Collecting a runtime summary of the fully instrumented application execution required an additional 50 min to create an experiment archive, unify definition identifiers and collate the severity values for the 42 call paths into a summary report after the application execution was completed. The application runtime itself was not appreciably dilated by the instrumentation and measurement and, since the estimated volume of trace data was less than 3MB per process, it was determined that no filtering would be required when tracing. Re-running the application collecting and analyzing 790GB of trace data took 150 min, of which dumping and loading the trace data took 7 min, and parallel replay of the entire trace to calculate the additional pattern metrics only 11 seconds. While 86 min was required to create and open one trace file per process, the latest version of Scalasca using a new parallel I/O library [2] that maps multiple logical files into a manageable number of physical files (in this case, 576, one per BG/P I/O node) reduces this time to 10 min and the actual trace I/O time to 4 min. Remaining bottlenecks in identifier unification and analysis report writing are currently being addressed.

The resulting 260MB analysis report presented for interactive exploration by the Scalasca analysis report explorer (Figure 4) shows the distribution of the *Late Receiver* metric for the call path to the `MPI_Send()`, where early sends are likely to block waiting for corresponding receives to commence. While the three-dimensional topology view derived from the Blue Gene/P physical torus reveals that a number of processes block almost four times longer than their peers, their distribution is difficult to discern. Presented using the application's two-dimensional virtual topology, however, a complex pattern is revealed comprising an interior rectangular region of processes with relatively high severity values and oblique lines of processes with lower severity values. This pattern results from the combination of diagonal 'sweeps' in each direction through the two-dimensional grid of application processes (primarily doing nearest neighbor exchanges) and imbalance in the preceding computation which recursively applies corrections to the calculated flux. While the *Late Receiver* waiting time consumed only a relatively small fraction of the total execution time (2%), the application spent more than a quarter (28%) in *Late Sender* wait states, where a receiver must wait for a message to arrive. *Late Sender* waiting time could be significantly reduced by adjusting SWEEP3D input parameters to use smaller grid blocks to increase pipelining concurrency, however, the intrinsic computational imbalance remains. While the imbalance is relatively small compared to the total sweep execution time (and therefore difficult to eliminate), its impact



```
0.00 Time
  2.94e7 Execution
    0.00 MPI
      0.00 Synchronization
        8.30 Collective
          563.45 Wait at Barrier
          1.96 Barrier Completion
      0.00 Communication
        3.90e5 Point-to-point
          9.75e7 Late Sender
          3.14e6 Late Receiver
        37622.61 Collective
          0.00 Early Reduce
          0.00 Early Scan
          8524.22 Late Broadcast
          1.89e7 Wait at N x N
          24.16 N x N Completion
      0.00 File I/O
      5.77e6 Init/Exit
  1.53e9 Overhead
  2.36e10 Visits
  8.85e5 Synchronizations
  0 Communications
    0 Point-to-point
      2.35e9 Sends
        3.53e9 Late Receivers
      3.53e9 Receives
        1.58e9 Late Senders
          7.71e8 Messages in Wrong
    1.06e7 Collective
  0 Bytes transferred
    0 Point-to-point
      4.51e13 Sent
      4.51e13 Received
    1.84e13 Collective
0.00        3.14e6 (2.03%)        1.55e8
```

```
0.00 driver
  0.00 task_init
    0.00 MPI_Init
    0.00 bcast_int
    0.00 barrier_sync
  0.00 read_input
    0.00 bcast_int
    0.00 bcast_real
  0.00 decomp
  0.00 inner_auto
    0.00 inner
      0.00 initialize
        0.00 initxs
        0.00 initsnc
          0.00 octant
        0.00 initgeom
      0.00 barrier_sync
        0.00 MPI_Barrier
      0.00 timers
      0.00 source
      0.00 sweep
        0.00 octant
        0.00 rcv_real
          0.00 MPI_Recv
        0.00 snd_real
          3.14e6 MPI_Send
      0.00 global_int_sum
        0.00 MPI_Allreduce
      0.00 flux_err
        0.00 global_real_max
          0.00 MPI_Allreduce
      0.00 global_real_sum
        0.00 MPI_Allreduce
  0.00 task_end
    0.00 MPI_Finalize
0.00        3.14e6 (100.00%)        3.14e6
```

```
0.00        0.00        100.00
4.75        3.14e6 (2.52e7%)        17.22
```

Late Receiver Time

Description:
A send operation may be blocked until the corresponding receive operation is called, and this pattern refers to the time spent waiting as a result of this situation.

Note that this pattern currently does not apply to nonblocking sends waiting in the corresponding completion call, e.g., `MPI_Wait`.

Unit:
Seconds

Diagnosis:
The MPI implementation may be working in synchronous mode by default, such that explicit use of asynchronous nonblocking sends can be tried. If the size of the message to be sent exceeds the available MPI internal buffer space then the operation will be blocked until the data can be transferred to the receiver: some MPI implementations allow larger internal buffers or different thresholds to be specified. Also consider the mapping of processes onto compute resources, especially if there are notable differences in communication time for particular processes, which might indicate longer/slower transmission routes or network congestion.

Parent:
MPI Point-to-point Communication Time

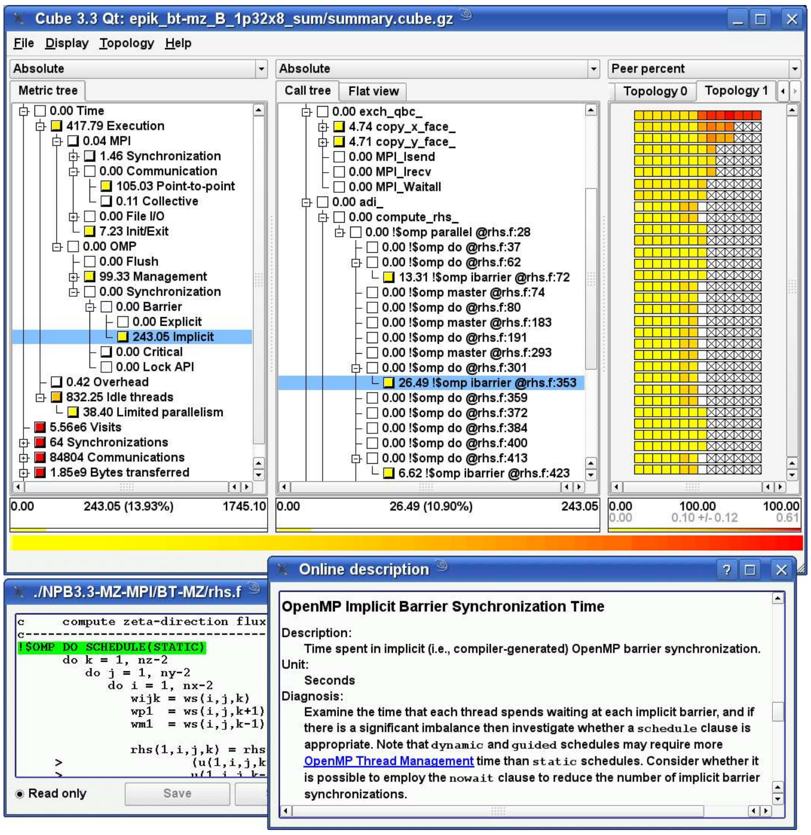
```
0.00        0.00        100.00
4.75        3.14e6 (2.52e7%)        17.22
```

Figure 4. Scalasca presentation of the trace-analysis report for a 294 912-process measurement of SWEEP3D on a 72-rack Blue Gene/P. *Late Receiver* time, when early sends are likely to be blocked waiting for corresponding receives to commence (as detailed in the online metric description window at lower left), amounts to more than 3 million seconds or 2.0% of the total execution time. With this selected in the metric tree pane (upper left), all of this time is attributed to the single call path ending with `MPI_Send()` in the call-tree pane (upper middle), and its distribution of values from 4.75 to 17.22 s for the 294 912 MPI processes is shown using the three-dimensional physical topology of Blue Gene/P (upper right) and the application's two-dimensional virtual 576 × 512 topology (lower right).

is magnified by the succession of communication sweeps which get disrupted and manifest as additional waiting times.

Scalasca 1.3 support for hybrid OpenMP/MPI measurement and analysis is demonstrated in Figure 5 by a summary analysis report from an execution of NAS NPB3.3 Block Triangular Multi-Zone (BT–MZ) benchmark [21] Class B in a Cray XT5 partition consisting of 32 compute nodes, each with two quad-core Opteron processors. One MPI process was started on each of the compute nodes, and OpenMP threads run within each SMP node. In an unsuccessful attempt at load balancing by the application, more than 8 OpenMP threads were created by the first 6 MPI ranks (shown at the top of the topological presentation in the right pane), and 15 of the remaining ranks only used 7 OpenMP threads. (Figure 5 presents OpenMP threads for each MPI process using a two-dimensional virtual topology, where void entries are shown hatched.) While the 38 s of *Limited parallelism* time for the unused threads/cores represents only 2% of the allocated compute resources, half of the total time is wasted by *Idle threads* while each process executes serially, including MPI operations done outside of parallel regions by the master thread of each process. Although the exclusive *Execution* time in serial computation is relatively well balanced on each OpenMP thread, the over-subscription of the first 6 compute nodes is manifested as excessive *Implicit Barrier Synchronization* time at the end of parallel regions (as well as additional OpenMP *Thread Management* overhead), and high *MPI Point-to-point Communication* time on the other processes is then a consequence of this. While creating different numbers of OpenMP threads for each MPI process can be effective for load-balancing in large SMP nodes, in cases like this with relatively small SMP nodes it is preferable to use the same number of threads for each node and avoid oversubscription of cores.

Beyond relatively simple benchmark kernels, Scalasca has also been successfully used to analyze and tune a number of locally important applications. Plasma physics and astrophysics simulations with the PEPC code, which employs a parallel tree algorithm for force computations, have been analyzed with Scalasca at different stages of porting and scaling on Blue Gene systems [7]. As the scale grew to employ thousands of processes, communication and load imbalance within the limited amount of processor memory became increasingly important. For example, after applying recently added iteration instrumentation capabilities to PEPC, we were able to observe the gradual formation of severe communication imbalances, requiring more innovative particle redistribution schemes to be developed and their efficiency evaluated. The XNS simulation of flow inside a blood pump, based on finite-element mesh techniques, was analyzed using up to 4096 processes on Blue Gene/L, and performance was improved more than four-fold after removing unnecessary synchronizations from critical scatter/gather operations and the system-provided `MPI_Scan()` routine [26]. Significant residual wait states diagnosed in the tuned version at 4096 processes even gave hope of further optimizations. On the MareNostrum blade cluster, the WRF2-NMM weather research and forecasting code was analyzed using 2048 processes, and occasionally seriously delayed exits from `MPI_Allreduce()` calls were identified that significantly degraded overall application performance (e.g. by inducing wait states in later communication operations), indicating a problem inside the MPI library or operating system interference [25]. In all these cases, the high-level call-path profile readily available from runtime summarization was key in identifying general performance issues that only manifested at the large scale, related to



```
0.00 Time
  417.79 Execution
    0.04 MPI
      1.46 Synchronization
      0.00 Communication
        105.03 Point-to-point
        0.11 Collective
      0.00 File I/O
      7.23 Init/Exit
    0.00 OMP
      0.00 Flush
      99.33 Management
      0.00 Synchronization
        0.00 Barrier
          0.00 Explicit
          243.05 Implicit
        0.00 Critical
        0.00 Lock API
  0.42 Overhead
  832.25 Idle threads
    38.40 Limited parallelism
5.56e6 Visits
64 Synchronizations
84804 Communications
1.85e9 Bytes transferred
```

```
0.00        243.05 (13.93%)        1745.10
```

```
0.00 exch_qbc_
  4.74 copy_x_face_
  4.71 copy_y_face_
  0.00 MPI_Isend
  0.00 MPI_Irecv
  0.00 MPI_Waitall
0.00 adi_
  0.00 compute_rhs_
    0.00 !$omp parallel @rhs.f:28
      0.00 !$omp do @rhs.f:37
      0.00 !$omp do @rhs.f:62
        13.31 !$omp ibarrier @rhs.f:72
      0.00 !$omp master @rhs.f:74
      0.00 !$omp do @rhs.f:80
      0.00 !$omp master @rhs.f:183
      0.00 !$omp do @rhs.f:191
      0.00 !$omp master @rhs.f:293
      0.00 !$omp do @rhs.f:301
        26.49 !$omp ibarrier @rhs.f:353
      0.00 !$omp do @rhs.f:359
      0.00 !$omp do @rhs.f:372
      0.00 !$omp do @rhs.f:384
      0.00 !$omp do @rhs.f:400
      0.00 !$omp do @rhs.f:413
        6.62 !$omp ibarrier @rhs.f:423
```

```
0.00        26.49 (10.90%)        243.05
```

```
0.00        100.00            100.00
0.00        0.10 +/- 0.12     0.61
```

```
c       compute zeta-direction flux
c----------------------------------
!$OMP DO SCHEDULE(STATIC)
      do k = 1, nz-2
         do j = 1, ny-2
            do i = 1, nx-2
               wijk = ws(i,j,k)
               wp1  = ws(i,j,k+1)
               wm1  = ws(i,j,k-1)

               rhs(1,i,j,k) = rhs
     >                  (u(1,i,j,k
```

OpenMP Implicit Barrier Synchronization Time

Description:
Time spent in implicit (i.e., compiler-generated) OpenMP barrier synchronization.
Unit:
Seconds
Diagnosis:
Examine the time that each thread spends waiting at each implicit barrier, and if there is a significant imbalance then investigate whether a `schedule` clause is appropriate. Note that `dynamic` and `guided` schedules may require more OpenMP Thread Management time than `static` schedules. Consider whether it is possible to employ the `nowait` clause to reduce the number of implicit barrier synchronizations.

Figure 5. Summary report presentation of a hybrid OpenMP/MPI NPB3.3 BT-MZ benchmark Class B execution on 32 Cray XT5 twin quad-core compute nodes, showing OpenMP *Implicit Barrier Synchronization* time for a parallel loop computing fluxes in the `compute_rhs()` routine (between lines 301 and 353 of source file rhs.f) broken down by thread in each of the 32 MPI processes.

performance problems that could be subsequently isolated and understood with targeted trace collection and analysis.

The SPEC MPI2007 suite of 13 benchmark codes from a wide variety of subject areas has also been analyzed with Scalasca with up to 1024 processes on an IBM SP2 p690+ cluster [20]. Problems were identified with several benchmarks that limited their scalability (sometimes to only 128 processes), such as distributing initialization data via broadcasts in 113.GemsFDTD and insufficiently large data sets for several others. Even those codes that apparently scaled well contained considerable quantities of waiting times indicating possible opportunities for performance and scalability improvement through more effective work distributions or bindings of processes to processors. 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 its execution behavior was repetitive, this allowed 2000 instead of the full 9000 timesteps to be specified for a representative shorter execution that could be traced and automatically analyzed.

HPC applications are typically tuned to use a large proportion of available memory, either for 'heroic' data sets or temporary storage of intermediate results, which naturally limits the memory remaining for trace collection buffers and other measurement data structures. In practice this has rarely been a constraint, as there are other compelling reasons to reduce trace sizes (both in memory and on disk) and thereby minimize trace I/O and analysis times.

While large-scale tests are valuable to demonstrate scalability, what has been more important is 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

Future enhancements will aim at further improving both the functionality and scalability of the Scalasca toolset. Completing support for OpenMP to eliminate the need for sequential trace analysis and to support more advanced features such as nested parallelism and tasking is a primary development objective. 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 analysis workflows including in-memory trace analysis that will allow us to handle even larger configurations. However, even then, the scalability of offline trace-analysis would still be limited in terms of the length of execution. Since obtaining full traces of realistic applications running for hours is rarely feasible without flushing memory buffers — which delays execution and possibly perturbs the measured behavior — tracing is usually restricted to intervals of limited duration. Based on preceding space-efficient characterizations of the often substantially time-dependent application behavior, we are therefore striving to offer more targeted trace collection mechanisms, reducing memory and/or disk space requirements while retaining the value of trace-based in-depth analysis. Specifically, we are working towards automatically identifying those execution phases that are either representative of larger groups of repetitive behaviors or cover singular but nonetheless pivotal performance phenomena and therefore justify more expensive trace analyses. At the same time, we are evaluating the feasibility of controlled on-line trace processing to be performed immediately after collective synchronization operations, which would have the potential to raise the limit on the number of events per process for at least some applications by allowing trace buffers to be periodically consumed.

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. W. Frings, F. Wolf, and V. Petkov. Scalable massively parallel I/O to task-local files. In *Proc. ACM/IEEE SC'09 Conference (Portland, OR, USA)*, November 2009.
3. 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.
4. 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, Amsterdam. IOS Press.
5. M. Geimer, S. S. Shende, A. D. Malony, and F. Wolf. A generic and configurable source-code instrumentation component. In *Proc. 9th Int'l Conf. on Computational Science (ICCS, Baton Rouge, LA, USA)*, volume 5545 of *Lecture Notes in Computer Science*, pages 696–705. Springer, May 2009.
6. 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.
7. M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proc. 13th European PVM and MPI Conference (EuroPVM/MPI, Bonn, Germany)*, Lecture Notes in Computer Science 4192, pages 303–312. Springer, September 2006.
8. M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. A scalable tool architecture for diagnosing wait states in massively-parallel applications. *Parallel Computing*, 35(7):375–388, 2009.
9. M.-A. Hermanns, M. Geimer, B. Mohr, and F. Wolf. Scalable detection of MPI-2 remote memory access inefficiency patterns. In *Proc. 16th European PVM and MPI Conference (EuroPVM/MPI, Espoo, Finland)*, volume 5759 of *Lecture Notes in Computer Science*, pages 31–41. Springer, September 2009.
10. Jülich Supercomputing Centre. *Scalasca scalable parallel performance analysis toolset documentation.* `http://www.scalasca.org/software/documentation`, last access 04.12.2009.
11. 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, Jülich, September 2005. Forschungszentrum Jülich.
12. 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(1):81–104, August 2002.
13. B. Mohr, A. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128, August 2002.
14. B. Mohr, B. J. N. Wylie, and F. Wolf. Performance measurement and analysis tools for extremely scalable systems. In *Proc. Int'l Supercomputing Conference (ISC'08, Dresden, Germany)*, June 2008.
15. W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
16. 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.
17. 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.
18. S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.
19. 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.
20. 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, pages 99–123. Springer, July 2008.
21. R. F. Van der Wijngaart and H. Jin. NAS Parallel Benchmarks, Multi-Zone versions. Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, USA, July 2003.
22. 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.

23. F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10–11):421–439, 2003.
24. 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 *Tools for High Performance Computing: Proc. 2nd HLRS Parallel Tools Workshop (Stuttgart, Germany)*, pages 157–168. Springer, July 2008.
25. B. J. N. Wylie. Scalable performance analysis of large-scale parallel applications on MareNostrum. In *Science and Supercomputing in Europe Report 2007*, pages 453–461. HPC-Europa Transnational Access, CINECA, Casalecchio di Reno (Bologna), Italy, 2008. ISBN: 978-88-86037-21-1.
26. 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.
27. 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*, 16(2-3):167–181, 2008.
28. 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.