Further Improving the Scalability of the Scalasca Toolset

Markus Geimer[1], Pavel Saviankou[1], Alexandre Strube[1,2], Zoltán Szebenyi[1,3], Felix Wolf[1,3,4], and Brian J.N. Wylie[1]

[1] Jülich Supercomputing Centre, 52425 Jülich, Germany
{m.geimer,p.saviankou,a.strube,z.szebenyi,b.wylie}@fz-juelich.de
[2] Universitat Autònoma de Barcelona, 08193 Barcelona, Spain
[3] RWTH Aachen University, 52056 Aachen, Germany
[4] German Research School for Simulation Sciences, 52062 Aachen, Germany
f.wolf@grs-sim.de

Abstract. Scalasca is an open-source toolset that can be used to analyze the performance behavior of parallel applications and to identify opportunities for optimization. Target applications include simulation codes from science and engineering based on the parallel programming interfaces MPI and/or OpenMP. Scalasca, which has been specifically designed for use on large-scale machines such as IBM Blue Gene and Cray XT, integrates runtime summaries suitable to obtain a performance overview with in-depth studies of concurrent behavior via event tracing. Although Scalasca was already successfully used with codes running with 294,912 cores on a 72-rack Blue Gene/P system, the current software design shows scalability limitations that adversely affect user experience and that will present a serious obstacle on the way to mastering larger scales in the future. In this paper, we outline how to address the two most important ones, namely the unification of local identifiers at measurement finalization as well as collating and displaying analysis reports.

Keywords: Scalasca, scalability.

1 Introduction

Driven by growing application requirements and accelerated by current trends in microprocessor design, the number of processor cores on modern supercomputers increases from generation to generation. With today's leadership systems featuring more than a hundred thousand cores, writing efficient codes that exploit all the available parallelism becomes increasingly difficult and requires adequate tool support for performance analysis. Unfortunately, increased concurrency levels impose higher scalability demands not only on applications but also on the software tools needed for their development. When applied to larger numbers of processors, familiar tools often cease to work in a satisfactory manner (e.g., due to serialized operations, escalating memory requirements, limited I/O bandwidth, or failed renderings).

K. Jónasson (Ed.): PARA 2010, Part II, LNCS 7134, pp. 463–473, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Scalasca [2] is an open-source toolset that can be used to analyze the performance behavior of parallel applications and to identify opportunities for optimization. Target applications include simulation codes from science and engineering written in C, C++ and Fortran and based on the parallel programming interfaces MPI and/or OpenMP. Scalasca has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but is also well suited for a wide range of small- and medium-scale HPC platforms. Scalasca combines runtime summaries suitable to obtain a performance overview with in-depth studies of concurrent behavior via event tracing. The traces are analyzed 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 serious challenges to achieving good performance. Thanks to a novel parallel trace-analysis scheme, the search can be performed even for very large numbers of cores. Internally, runtime summarization and tracing are tightly integrated, allowing the user to switch between the two modes via environment variables and even to apply them simultaneously.

Although Scalasca's scalable design already facilitated performance analyses of application runs on a Blue Gene/P system with 294,912 (288k) cores [6], the architecture underlying version 1.3.0 (released March 2010) still shows scalability limitations, which primarily occur (i) during the unification of local identifiers at measurement finalization and (ii) while collating and displaying analysis reports. Both limitations adversely influence user experience in the form of either increased time needed for performance-data acquisition or prolonged response times during the interactive exploration of analysis results. Since they will present serious obstacles on the way to mastering higher scales in the future, we outline in this paper how they can be effectively addressed to ensure scalability even when hundreds of thousands of cores are employed. In the remainder of the paper, we explain each of the two challenges in more detail along with the progress achieved so far, followed by related work and an outlook on what still needs to be done to match our objective of substantially increasing Scalasca's scalability.

2 Unification of Local Identifiers

In Scalasca, event data measured as part of the traces 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 numerical identifiers, while the objects themselves are defined separately. However, to establish a global view of the program behavior during analysis, these identifiers must be consistent across all processes. Unfortunately, generating global identifiers purely locally as a hash function of some definition key would pose the danger of global conflicts, which are very hard to resolve. For this reason, each process may use a different local identifier to denote the same object. However, ultimately a global set of unique object definitions must be created and local identifiers mapped onto global identifiers in a consistent manner.

This procedure, which is called *unification* and which requires exchanging and comparing definitions among different processes, is performed during application finalization to avoid perturbation during measurement. Since runtime summarization and tracing share the same set of definitions, unification is also needed in summary mode. Definitions always refer to processes with potentially multiple threads, which is why both the previous and the new unification algorithms equally apply to pure MPI as well as hybrid OpenMP/MPI applications.

Although Scalasca's current unification algorithm already takes advantage of message communication to facilitate the efficient exchange of object definitions and the generation of local-to-global identifier mapping tables, it is still predominantly sequential, posing a serious scalability limitation. To overcome this situation, this sequential step was parallelized using a hierarchical algorithm so that it can be efficiently performed for codes running on hundreds of thousands of cores. It consists of the following three steps:

- generation of a unified set of global definitions,
- generation of local-to-global identifier mappings for each process, and
- writing the global set of definitions as well as the identifier mappings to disk.

Note that the last step is only required in tracing mode, since the identifier mapping can already be applied at run time in summarization mode. In the following paragraphs, all three steps will be described in more detail.

2.1 Definition Unification

Unifying object definitions is a data-reduction procedure that combines the local definitions from each process by first removing duplicates and then assigning a unique global identifier to each of the remaining definitions. In the new hierarchical scheme, this is done in several iterations as can be seen in Figure 1. During the first iteration, processes with odd rank numbers send their definitions to their neighbor processes with even rank numbers, which unify them with their own definitions. During subsequent iterations, these partially unified definitions are exchanged in a similar way, doubling the rank offset between sender and receiver in each step. In the end – after $\lceil \log_2 P \rceil$ iterations – the unified set of global definitions is available at rank zero.

During this step, it is essential to use efficient data structures and algorithms for identifying whether a particular definition object has already been defined or not. Depending on the types of the definition objects and their expected number, we use hash tables (e.g., for character strings), trees (e.g., for call paths), and vectors (e.g., for Cartesian topology definitions). Moreover, definition objects which are known to be unique across all processes, such as so-called "locations" referring to a thread of execution or Cartesian topology coordinates, are simply aggregated to avoid unnecessary search costs.

Note that an important prerequisite for using the hierarchical algorithm is that the semantics of the individual attributes of a definition object remain stable, which unfortunately was not the case for Cartesian topology definitions

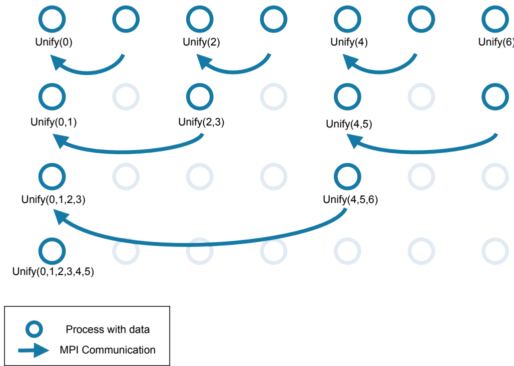


Fig. 1. Hierarchical definition unification algorithm

in the original scheme. Here the topology identifier assigned by the measurement system was either the identifier of the associated communicator for MPI Cartesians or a special constant for user-defined and hardware topologies. These would have been mapped to identifiers in the range $[0, n-1]$ during the first iteration of the unification, communicator or the special constant and creating a need for a different unification algorithm during subsequent iterations. Since the communicator identifier is also available via a second attribute, this issue could be resolved by having the measurement system directly assign local topology identifiers in the range $[0, n-1]$. As a positive side effect, this change now also allows us to handle more than one user-defined or hardware topology.

2.2 Generation of Local-to-Global Identifier Mappings

With the global set of definitions at hand, a translation table, in the following called *mapping*, can be created that maps the local identifiers used by each process onto the identifiers of the global definitions. This is done by broadcasting the unified global definitions from rank zero to all other processes, which subsequently compare their local definitions with the global ones to determine the identifier mapping.

Here it is crucial to exclude the above-mentioned definition objects from the broadcast that are unique to each process, since their data volume may dominate the overall size of the global definitions even at relatively small scales (512–1024 processes) and lead to prohibitively large data sets to be transferred at larger scales. However, unique identifiers for location objects and an associated mapping are still required. Due to the deterministic nature of the hierarchical unification scheme presented in Section 2.1, the global location identifiers can be locally reconstructed by determining the identifier offset via an exclusive prefix

sum over the number of locations per process (emulated using `MPI_Scan` and a local subtraction for compatibility with non-MPI 2.x compliant MPI implementations).

2.3 Writing Definition Data and Mappings to Disk

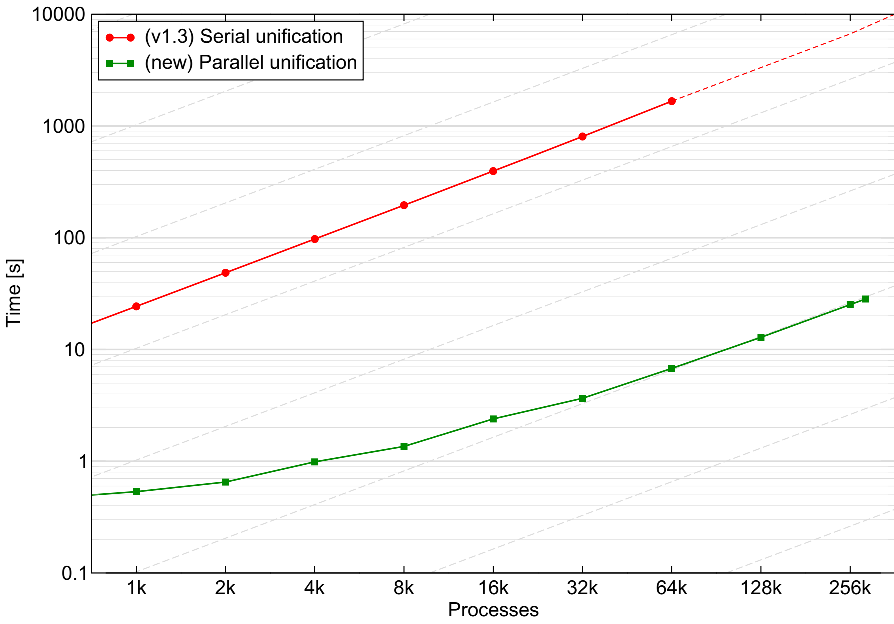
To avoid expensive rewriting, traces still contain the local identifiers when they are written to disk. Therefore, the per-process identifier mappings need to be stored on disk as well if tracing mode is configured, from where they are later retrieved to correctly resolve the still existing local-identifier references in the traces before they are processed by Scalasca's parallel trace analyzer. Currently, all mappings are sequentially written to a single mapping file, with the file offset for each rank stored in the global definitions file.

Ideally, all processes would write their information in parallel to disjoint sections of the single mapping file, however, the amount of data per process is rather small so that lock conflicts on the file-system block level would significantly degrade I/O performance in this scenario. Therefore, the mapping information is gathered in chunks of 4 MB on a small set of processes, with the root process within each group defined as the first rank providing data for a particular chunk being gathered. Since there is no such "multi-root gather" operation provided by the MPI standard, it has been implemented using point-to-point messages and a hierarchical gather algorithm, very similar to the one used for the definition unification.

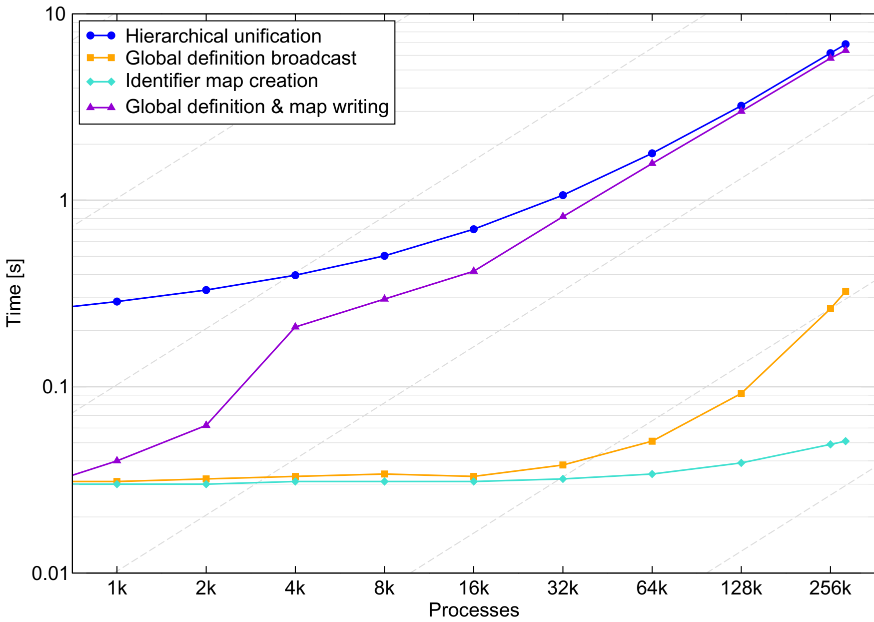
Using parallel I/O from this small set of gather processes was evaluated on the IBM Blue Gene/P system Jugene at Jülich Supercomputing Centre using a GPFS parallel file system. However, we found that the setup costs grew significantly when more than one I/O node was involved, which could not be amortized while writing the relatively small amount of data. Our current solution therefore lets rank zero write the mappings incrementally after receiving the 4 MB chunks already collected during the multi-root gather operations. In addition, rank zero gathers the map file offsets for each rank and writes them to the global definitions file. The offsets are again calculated using an exclusive prefix sum over the amount of mapping data generated by each process.

2.4 Experimental Results

Figure 2(a) shows a comparison of the original serial version of the unification algorithm and the new parallel algorithm for the definition data produced by a fully compiler-instrumented binary of the SMG2000 benchmark code [1] at various scales up to 288k processes, measured on the IBM Blue Gene/P system Jugene at the Jülich Supercomputing Centre with no run-time filtering applied. Although still exhibiting a similar scaling behavior, it can be seen that the new algorithm shows a remarkable improvement over the previous sequential version, reducing the time to unify the definitions at large scales by several orders of magnitude.



(a)



(b)

Fig. 2. Execution times of the new parallel and the previous serial unification schemes (a) as well as a breakdown of the individual steps of the parallel scheme (b) for the definition data of the SMG2000 benchmark code. (Note the different y-axis scales.)

Figure 2(b) shows a detailed breakdown of the unification runtime into the different steps involved, allowing the scalability of each step to be individually judged. As could be expected, the time to write the global definition data and mapping information is growing linearly at larger scales. This also applies to the hierarchical unification, where the scale-dependent data (i.e., the definition objects which are unique for each process as well as MPI communicator definitions) start to dominate the overall amount of data that is collected. Although almost all of this kind of data is not included in the broadcast and hence not considered during the identifier map creation, this does not apply to the definitions of communicators.

Currently, Scalasca encodes each communicator as a bitstring, with the i-th bit indicating whether the global rank i is part of the communicator or not. Therefore, the size of a communicator definition is linearly dependent on the number of processes, causing also communicator definitions to become a dominant part of the overall data volume sent during the broadcast and processed during the identifier map creation at some point. This suggests that a revised representation of communicators is required to further improve the scalability of these two steps (as well as the hierarchical unification). The design of a more space-efficient distributed scheme to record the constituency of a communicator is already in progress.

3 Collating and Displaying Analysis Reports

At the end of runtime summary measurements and after automated parallel trace analysis, Scalasca produces intermediate analysis reports from the unified definitions and the metric values for each call path collated from each process. The definitions of the measured and analyzed metrics, program call tree, and system configuration constitute metadata describing the experiment, where only the latter vary with the number of processes and threads. On the other hand, the amount of metric value data increases linearly. Intermediate analysis reports are sequentially post-processed to derive additional metrics and create a structured metric hierarchy prior to examination with textual or graphical tools.

Scalasca 1.3 saves analysis reports as single files using XML syntax, where ‘exclusive’ metric values are stored for each call path (explained in Section 3.3). This means that exclusive values first have to be calculated and then the numeric values have to be converted to formatted text when writing. Since values which are not present in reports default to zero when reports are read, a ‘sparse’ representation can be exploited which avoids writing vectors consisting entirely of zeroes. Furthermore, the final report is typically compressed when writing, which is advantageous in faster writing (and reading) time as well as much smaller archival size. To calculate inclusive metric values, all of the exclusive metric values must be read and aggregated.

Scaling of Scalasca analysis report collation time and associated analysis report explorer GUI loading time and memory requirements for Sweep3D trace analyses on Jugene [6] are shown in Figure 3.

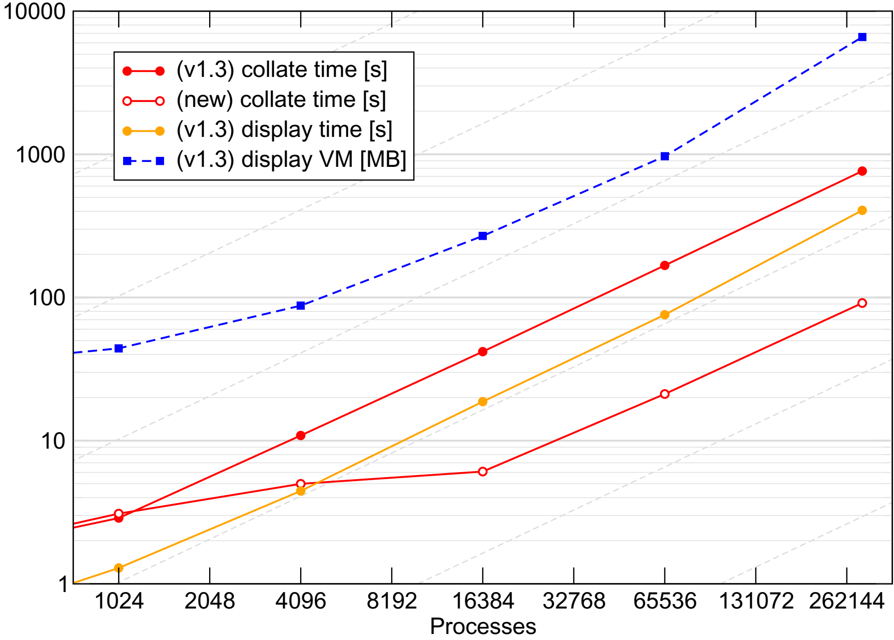


Fig. 3. Scalasca Sweep3D trace analysis report collation time, with associated loading time and memory requirements for the analysis report explorer GUI

3.1 Storing Metric Values in Binary Format

To reduce the collation time, we designed a new file format for storing experiment data, keeping the XML format essentially unchanged for storing metadata, while introducing a new binary format and file organization for storing metric values. In this way, third party tools such as TAU [5] that implement their own reader and/or writer for Scalasca analysis reports can keep their implementations for the more complicated metadata part of the report and only re-implement readers for the metric index and value files.

Figure 3 shows that when using the new format at smaller scales the collation time is dominated by writing the metadata, whereas beyond 16 thousand processes the writing time scales linearly with the amount of metric values to be written (regardless of the format). As in Section 2.3, writing of metric values in parallel from a configurable number of compute nodes was investigated, and again, best performance was obtained using a single writer. Collation time for the 288k-process analysis report is reduced from 13 minutes to 90 seconds. The sizes of both old and new analysis report formats grow linearly with the number of processes (as well as with the number of metrics and measured call paths). While the uncompressed intermediate reports are comparable in size between both format versions, the currently uncompressed set of binary files is typically four times larger than the (compressed) XML report, in this case 1530 MB vs.

404 MB for the 288k-process trace analysis report. When the file-system bandwidth is low compared to the processor speed, however, the performance gained from compression when writing intermediate or post-processed reports may justify the effort required to run the compression algorithm. For this reason, we are contemplating a configurable solution with optional compression.

3.2 File Organization

Instead of writing the entire report to a single XML file as before, we split the file into an XML part for metadata and a binary part for metric values. In anticipation of the dynamic-loading capabilities described later, each metric is stored in a separate file. This file is accompanied by an index file that specifies the data layout, allowing a sparse data representation if needed. A trace analysis report such as the ones considered here now consists of 197 files in total.

3.3 Dynamic Loading

Figure 3 shows that the time to load an entire analysis report and the amount of memory required increase slightly worse than linearly with the number of processes. For the 288k-process Sweep3D trace analysis report, loading takes 7 minutes on the 4.2 GHz Power6 Jugene BG/P front-end system and requires 6.5 GB of memory. (Naturally, a 64-bit version of the GUI and other report tools is necessary for such large amounts of analysis data.) Although paging to disk is fortunately avoided here, each GUI interaction still requires several minutes, seriously impairing interactive analysis report exploration. Command-line utilities to post-process analysis reports are found to require up to 15 GB memory and more than 30 minutes execution time. Since the binary analysis reports are much larger than the compressed XML reports, utilities using the new format are actually 10% slower when reading the entire report.

The solution to these problems is to avoid reading all the data into memory at once. With the new format, it is now possible to read any part of the data on-demand. In a typical usage scenario, the GUI would initially show only the root of the call tree (i.e., main) plus associated metric values. As soon as the user expands the root node, only the data of its direct children are loaded. The children of a node can easily be stored in consecutive order, in which case this will likely require just a single file system access. Of course, the indices for the metrics are small enough to be kept in memory, allowing for efficient data lookup.

Another major source of prolonged response times is that metric values for individual call-tree nodes (i.e., call paths) are stored with exclusive semantics. Exclusive means that the value stored for a given call-tree node refers to that node only – its children excluded – as opposed to an inclusive value, which is the sum of the exclusive values in the subtree anchored at the node. Typically we want to display both exclusive and inclusive values for a given node and if we store one, we can calculate the other. Calculating inclusive values from exclusive ones is very expensive, requiring a complete traversal of the corresponding subtree and adding up all values after loading them into memory, whereas

converting from inclusive to exclusive is cheap. One just subtracts the values of the children from the value of the current node. In spite of losing some of the sparsity that can be exploited to save disk space, inclusive semantics are preferable with dynamic-loading. This would reduce the memory usage of the GUI to just the data being currently displayed on screen (plus optionally some additional caching). Of course, incremental reading can be easily extended to incremental rewriting, limiting the memory footprint of most post-processing tools to basically a single row of data at a time. At the time of writing, a quantitative comparison was not yet available.

3.4 Index Structures for Sparse Data

While a few of the metrics such as time and the number of times a call path has been visited (and optional hardware counter metrics) are dense, having non-zero values for each call path, many of the metrics are very sparse. For example, MPI point-to-point communication time has non-zero values only for the corresponding MPI calls (and their parent call paths when stored with inclusive semantics). For these metrics, we use a sparse representation, that is, we refrain from writing the rows of data that would only contain zeros in this metric. To keep track of which rows are written and allow for efficient lookup of the data straight from the binary file, the index file contains a list of the identifiers of the call paths whose data rows were actually stored. Keeping this index in memory, we can find the appropriate file offset for any call path in logarithmic time. Utilities which process analysis reports and change call path identifiers, such as remap and cut, therefore need to update the primary XML metadata, any secondary indices that refer to modified call-path identifiers, and potentially also the files containing the associated metric data.

4 Related Work

Obviously, the problem of unifying process-local definitions is not specific to the Scalasca toolset. For example, the VampirTrace measurement library [3] performs the unification as a serial post-processing step, which in addition also rewrites the generated trace files using the global identifiers. A second example is the TAU performance system [5], which serially unifies call-path information on-the-fly while loading profile data into the graphical profile browser ParaProf. Recently, a hierarchical unification algorithm for call paths has been implemented to support on-line visualization of so-called snapshot profiles. Treating unification as a global reduction problem puts it into a larger category of global reduction operations used in parallel performance analysis, many of them applied online [4].

Each performance tool typically has its own native format for profile data. VampirTrace produces OTF profiles consisting of separate binary files for each MPI process rank and several additional metadata files, which are subsequently integrated into a plain text profile. TAU also produces individual textual profiles

for each MPI rank by default, and can then combine these into a single-file packed binary format, or alternatively, it can concatenate the individual profiles from each process rank at the end of measurement into a single textual file for each performance metric. TAU's profile viewer can read all of these formats and many more, including Scalasca integrated XML profiles and individual gprof textual profiles.

5 Conclusion and Outlook

In this paper, we showed how to address the most serious impediments to achieving further scalability in Scalasca. The previously serial unification of local identifiers created to reference mostly global objects was parallelized using a hierarchical reduction scheme, accelerating the procedure by almost a factor of 250 on 64k cores (and an estimated factor of about 350 on 288k cores) on an IBM Blue Gene/P. Subsequent tests on other large-scale HPC systems, such as Cray XT, showed similar benefits. Therefore, the parallel unification was integrated into the Scalasca measurement system and is already part of the latest 1.3.2 release (November 2010). Further improvements can be achieved by revising the handling and representation of MPI communicators as well as optimizing the identifier mappings, which still contain a significant amount of redundancy.

A speedup of more than 7 was observed for collating and writing analysis reports on 288k cores after replacing the XML file format used to store metric values with a binary alternative. Further optimizations of the file layout and the underlying data model as well as dynamic loading capabilities are likely to improve the interactive response times of the report explorer GUI.

References

1. Accelerated Strategic Computing Initiative: The ASC SMG2000 benchmark code (2001), `http://www.llnl.gov/asc/purple/benchmarks/limited/smg/`
2. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
3. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008)
4. Roth, P.C., Arnold, D.C., Miller, B.P.: MRNet: A software-based multicast/reduction network for scalable tools. In: Proc. Supercomputing Conference, SC 2003, Phoenix, AZ, USA (November 2003)
5. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–331 (2006)
6. Wylie, B.J.N., Böhme, D., Mohr, B., Szebenyi, Z., Wolf, F.: Performance analysis of Sweep3D on Blue Gene/P with the Scalasca toolset. In: Proc. International Parallel & Distributed Processing Symposium, Workshop on Large-Scale Parallel Processing IPDPS–LSPP, Atlanta, GA, USA. IEEE Computer Society (April 2010)