Scalable Collation and Presentation of Call-Path Profile Data with CUBE

Markus Geimer[1], **Björn Kuhlmann**[1,3], **Farzona Pulatova**[1,2], **Felix Wolf**[1,3], **and Brian J. N. Wylie**[1]

[1] Forschungszentrum Jülich,
John von Neumann Institute for Computing, 52425 Jülich, Germany
E-mail: {m.geimer, f.wolf, b.wylie}@fz-juelich.de, bjoern.kuhlmann@sap.com

[2] University of Tennessee,
Innovative Computing Laboratory, Knoxville, TN 37996-3450, USA
E-mail: farzona@gmail.com

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

Developing performance-analysis tools for parallel applications running on thousands of processors is extremely challenging due to the vast amount of performance data generated, which may conflict with available processing capacity, memory limitations, and file system performance especially when large numbers of files have to be written simultaneously. In this article, we describe how the scalability of CUBE, a presentation component for call-path profiles in the SCALASCA toolkit, has been improved to more efficiently handle data sets from thousands of processes. First, the speed of writing suitable input data sets has been increased by eliminating the need to create large numbers of temporary files. Second, CUBE's capacity to hold and display data sets has been raised by shrinking their memory footprint. Third, after introducing a flexible client-server architecture, it is no longer necessary to move large data sets between the parallel machine where they have been created and the desktop system where they are displayed. Finally, CUBE's interactive response times have been reduced by optimizing the algorithms used to calculate aggregate metrics. All improvements are explained in detail and validated using experimental results.

1 Introduction

Developing performance-analysis tools for applications running on thousands of processors is extremely challenging due to the vast amount of performance data usually generated. Depending on the type of performance tool, these data may be stored in one or more files or in a database and may undergo different processing steps before or while they are shown to the user in an interactive display. Especially when some of these steps are carried out sequentially or make use of shared resources, such as the file system, the performance can be adversely affected by the huge size of performance data sets. Another scalability limit is posed by memory capacity, which may conflict with the size of the data to be held at some point in the processing chain. As a consequence, performance tools may either fail or become so slow that they are no longer usable. One aspect where this is particularly obvious is the visual presentation of analysis results. For example, a viewer may prove unable to load the data or long response times may compromise interactive usage. And even when none of the above applies, limited display sizes may still prevent a meaningful presentation.

In this article, we describe how CUBE[1], a presentation component for call-path profiles that is primarily used to display runtime summaries and trace-analysis results in the SCALASCA performance tool set[2] for MPI applications, has been enhanced to meet the requirements of large-scale systems. While transitioning to applications running on thousands of processors, scalability limitations appeared in the following four areas:

1. Collation of data sets: Originally, the data corresponding to each application process was written to a separate file and subsequently collated using a sequential program – a procedure that performed poorly due to the large numbers of files being simultaneously created and sequentially processed.

2. Copying data sets between file systems: When the user wanted to display the data on a remote desktop, the relatively large data sets first had to be moved across the network.

3. Memory capacity of the display: As the data sets grew larger with increasing processor counts, they could no longer be loaded into the viewer.

4. Interactive response times: The on-the-fly calculation of aggregate metrics, such as times accumulated across all processes, consumed increasingly more time.

To overcome these limitations, we have redesigned CUBE in various ways. The elements of our solution include (i) a parallel collation scheme that eliminates the need to write thousands of files, (ii) a client-server architecture that avoids copying large data sets between the parallel machine and a potentially remote desktop, (iii) an optimization of the display-internal data structures to reduce the memory footprint of the data, and (iv) optimized algorithms for calculating aggregate metrics to improve interactive response time. The overall result is substantially increased scalability, which is demonstrated with realistic data sets from target applications running on up to 16,384 processors.

The article is organized as follows: In Section 2, we briefly review the CUBE display component. Then, in Section 3, we describe the parallel collation scheme and how it is used in SCALASCA, before we explain the client-server architecture in Section 4. The optimized internal data structures along with the improved aggregation algorithms are presented in Section 5. Finally, we conclude the paper and point to future enhancements in Section 6. Measurement results appear in the text along with the contents they refer to.

2 CUBE

The CUBE (CUBE Uniform Behavioral Encoding) display component is a generic graphical user interface for the presentation of runtime data from parallel programs, which may include both performance data but also data on, for example, runtime errors. It has been primarily designed for use in SCALASCA, but is also used in combination with the TAU performance tool suite[3] and the MPI error detection tool MARMOT[4].

Data model. CUBE displays data corresponding to a single run of an application, which is called an *experiment*. The internal representation of an experiment follows a data model consisting of three dimensions: a metric dimension, a call-tree dimension, and a system

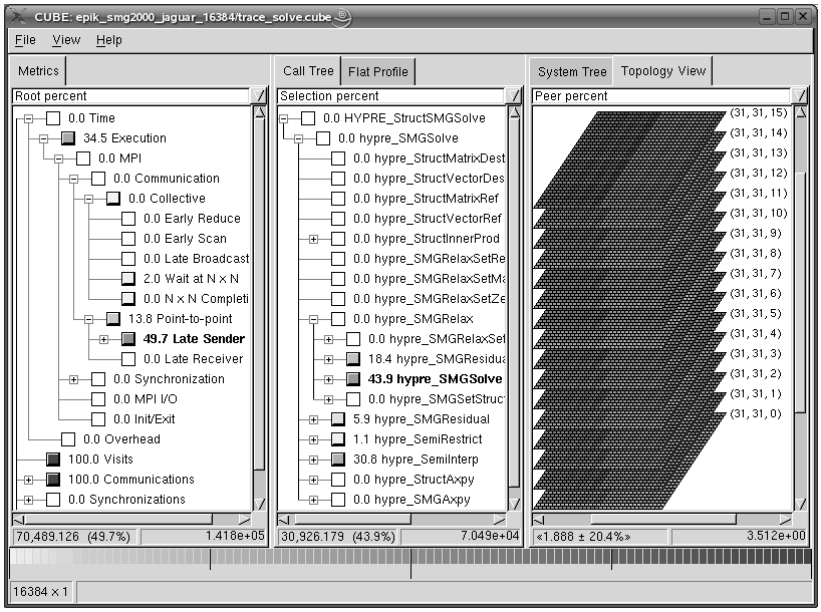

Figure 1. CUBE display showing a SCALASCA trace-analysis report for the ASC SMG2000 benchmark on 16,384 processors. The display shows the hierarchy of performance metrics (left pane), the call tree of the program (middle pane), and the application processes arranged in a three-dimensional virtual topology (right pane).

dimension. Motivated by the need to represent performance behavior on different levels of granularity as well as to express natural hierarchical relationships among metrics, call paths, or system resources, each dimension is organized in a hierarchy.

The metric hierarchy is intended to represent metrics, such as times or events, and may provide more general metrics at the top (e.g., execution time) and more specialized metrics closer to the bottom (e.g., communication time). The call-tree hierarchy contains the different call paths a program may visit during execution (e.g., main() $\rightarrow$ foo() $\rightarrow$ bar()). In the context of pure message-passing applications, the system hierarchy consists of the three levels machine, (SMP) node, and process. However, in general it can include an additional thread level to represent analysis results from multi-threaded programs. Besides the hierarchical organization, the processes of an application can also be arranged in physical or virtual process topologies, which are part of the data model. A severity function determines how all the tuples (metric m, call path c, process p) of an experiment are mapped onto the accumulated value of the metric m measured while the process p was executing in call path c. Thus, an experiment consists of two parts: a definition part that defines the three hierarchies and a data part representing the severity function.

File format. CUBE experiments can be stored using an XML file format. A file representing a CUBE experiment consists of two parts: the definitions and the severity function values. The severity values are stored as a three-dimensional matrix with one dimension for the metric, one for the call path, and one for the process. CUBE provides a C++ API for creating experiments and for writing them to or loading them from a file.

Display. CUBE's display consists of three tree browsers representing the metric, the program, and the system dimension from left to right (Figure 1). Since the tree representation of the system dimension turned out to be impractical for thousands of processes, CUBE provides a more scalable two- or three-dimensional Cartesian grid display as an alternative to represent physical or virtual process topologies.

Essentially, a user can perform two types of actions: selecting a tree node or expanding/collapsing a tree node. At any time, there are two nodes selected, one in the metric tree and another one in the call tree. Each node is labeled with a severity value (i.e., a metric value). A value shown in the metric tree represents the sum of a particular metric for the entire program, that is, across all call paths and the entire system. A value shown in the call tree represents the sum of the selected metric across the entire system for a particular call path. A value shown in the system tree or topology represents the selected metric for the selected call path and a particular system entity. To help identify high severity values more quickly, all values are ranked using colors. Due to the vast number of data items, the topological display shows only colors. Exploiting that all hierarchies in CUBE are inclusion hierarchies (i.e., that a child node represents a subset of the parent node), CUBE allows the user to conveniently choose between inclusive and exclusive metrics by collapsing or expanding nodes, respectively. Thus, the display provides two aggregation mechanisms: aggregation across dimensions (from right to left) by selecting a node, and aggregation within a dimension by collapsing a node.

3 Parallel Collation of Input Data

In SCALASCA, the trace analyzer, a parallel program in its own right, scans multiple process-local trace files in parallel to search for patterns of inefficient behavior. During the analysis, each analysis process is responsible for the trace data generated by one process of the target application. In the earlier version, at the end of the analysis, all analysis processes created their own CUBE file containing the process-local analysis results, which then had to be collated into a single global result file in a sequential postprocessing step. This initial approach had a number of disadvantages. First, every local result file contained exactly the same definition data, causing the analyzer to write large amounts of redundant information. Second, sequentially collating the local results scaled only linearly with the number of processes. This was aggravated by the fact that first writing the local results to disk and then reading them back for collation during the postprocessing phase incurred expensive but actually unnecessary I/O activity.

To overcome this situation, we devised a new mechanism that lets every analysis process send its local result data across the network to a single master process that writes only a single CUBE file containing the collated results from all processes. This has the advantage that creating a large number of intermediate files and writing redundant definition data can be avoided. Although this idea seems trivial, it has to cope with the challenge that the size of the global data set may easily exceed the memory capacity of the collating master. Therefore, to ensure that the master never has to hold more data at a time than its capacity allows, the data is incrementally collected in very small portions and immediately written to the global result file as it arrives.

The CUBE data model stores the severity function in a three-dimensional matrix indexed by the triple (metric, call path, process). Accordingly, our new collation algorithm

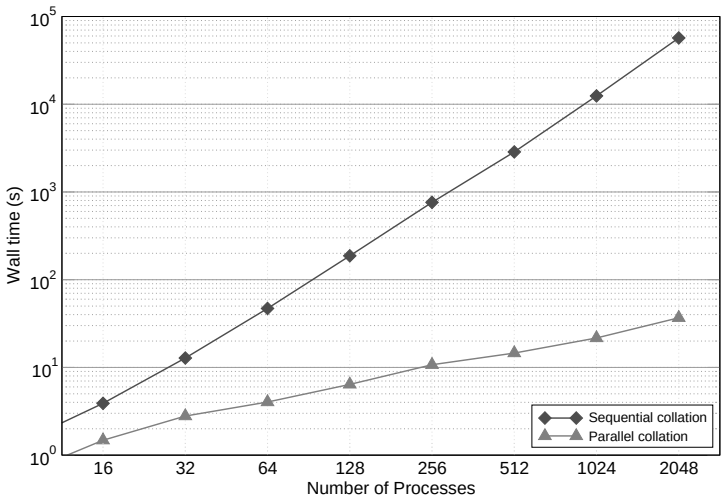

Figure 2. Comparison of trace-analysis result collation times for the ASC SMG2000 benchmark code on Blue Gene/L. The sequential collation was carried out on the front-end node, whereas the parallel collation was carried out on compute nodes.

consists of two nested loops, where the outer one iterates over all defined performance metrics and the inner one iterates over individual call paths. Since we assume that each analysis process stores only local results, the local process dimension for a given (metric, call path) combination consists only of a single entry. During an iteration of the inner loop, just this single value is collected from all analysis processes using an MPI gather operation. Then, the master process immediately collates all these values into the full (metric, call path) submatrix and writes it to the global file before the next iteration starts. In this way, even for 100,000 processes not more than 1 MB of temporary memory is required at a time. To prevent the master process from running out of buffer space when using an MPI implementation with a non-synchronizing gather, we placed a barrier behind each gather operation. To support the incremental write process, a special CUBE writer library is provided, which is implemented in C to simplify linkage to the target application if result files have to be written by a runtime library.

Figure 2 compares analysis result collation times of the initial sequential version and the new parallel version for the ASC SMG2000 benchmark code[5] running on up to 2,048 processes on Blue Gene/L. As can be seen, the sequential collation approach becomes more and more impractical at larger scales, whereas the improved algorithm scales very well. Even for 16,384 processes, the parallel scheme took less than five minutes.

4 Client-Server Architecture

To avoid copying potentially large CUBE data sets from the supercomputer where they have been generated across the network to a remote desktop for visualization, the previously monolithic CUBE display was split into a client and a server part. In this arrangement, the server is supposed to run on a machine with efficient access to the supercomputer's

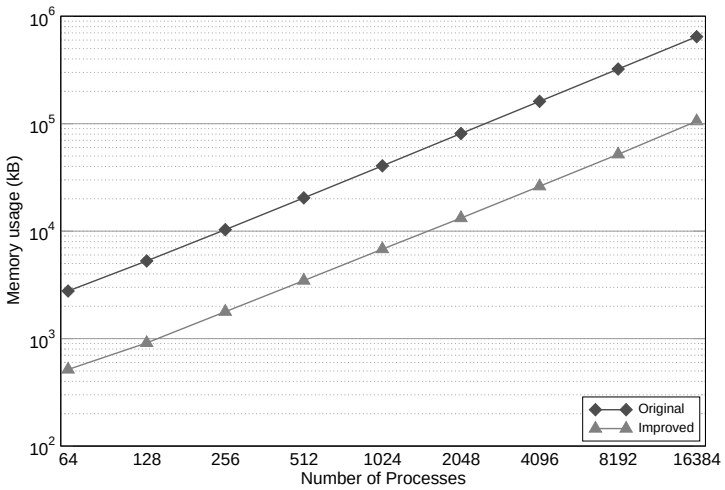

Figure 3. Comparison of the memory footprint of CUBE on a Linux workstation for trace analyses obtained from runs of the ASC SMG2000 benchmark code at a range of scales using the original (map) and the revised (vector) internal data representation.

file system (e.g., the front-end node), while the client, a lightweight display component, runs on the user's desktop system, querying the required data from the server over a BSD socket connection and displaying it without substantial further processing. The data is transferred in relatively small chunks corresponding to the values needed to populate one tree in the display (e.g., the metric tree or the call tree). Whenever the user performs an action requiring the update of one or more trees, the server performs all the necessary calculations including the calculation of aggregate metrics before sending the results to the client. To ensure security of the transmission, we rely on tunneling the connection through SSH, which is widely available and usually works smoothly even with restrictive firewall configurations.

In addition to solving the problem of copying large amounts of data, the server can also take advantage of a more generous hardware configuration in terms of processing power and memory available on the machine where it is installed. In this way, it becomes possible to hold larger data sets in memory and to perform the relatively compute-intensive calculation of aggregate metrics quicker than on the user's desktop. In a later stage, even a moderate parallelization of the server is conceivable.

The underlying idea of separating the display from the actual processing of the data has also been used in the design of Vampir Server[6], where the trace data is accessed and prepared by a parallel server program, before it is presented to the user in a pure display client.

5 Optimized Internal Data Structures and Algorithms

Since a substantial number of entries in the three-dimensional severity matrix are usually zero (e.g., no communication time in non-MPI functions), the original C++ GUI imple-

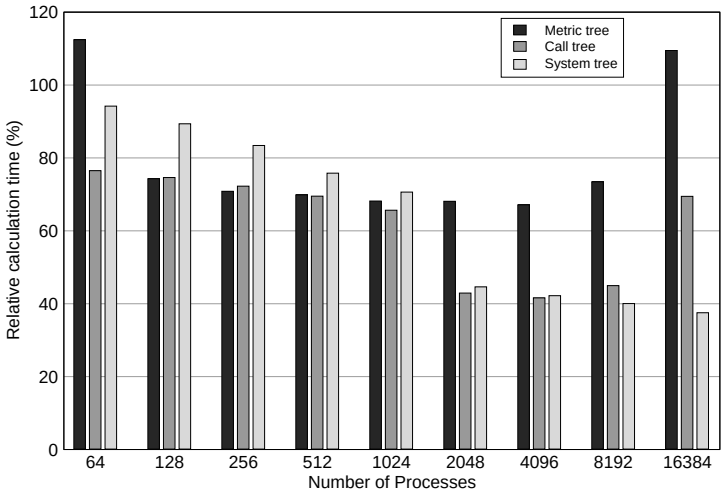

Figure 4. Calculation times of the improved aggregation algorithm for the three different hierarchies in relation to the original implementation (=100%) for trace analyses obtained from runs of the ASC SMG2000 benchmark code at a range of scales. The measurements were taken on a Linux workstation.

mentation used the associative STL container class `std::map` to store only the non-zero entries in the manner of a sparse matrix. The data structure employed a three-level nesting of maps using metric, call-path, and process pointers as keys.

However, experiments showed that in practice the third level, which stores the severity values of all processes for a particular tuple (metric, call path), is usually densely populated. Because STL maps are often implemented using some form of self-balancing binary search tree, this observation implied an opportunity for a substantial reduction of the memory overhead and at the same time for an improvement of the access latency for individual entries. Hence, the lowest nesting level was replaced by `std::vector`. However, this change required the global enumeration of all processes so that the process number could be used as vector index.

Figure 3 compares the memory consumption of the original implementation to the consumption of the revised version on a Linux workstation for trace-analysis data sets obtained from runs of the ASC SMG2000 benchmark code at various scales (similar results have been obtained for other data sets). As can be seen, changing from map to vector at the process level led to a significant reduction of the memory footprint. Depending on the data set and the number of processes, a reduction by factors between three and six could be observed. As a result, the CUBE viewer is now able to display analysis results at much larger scales.

In the course of the above modification, we enumerated also metrics and call paths to improve the performance of some of the algorithms used to calculate aggregate metrics, such as the accumulated time spent in a specific call path including all its children. In particular, the calculation of the inclusive and exclusive severity values for the metric, call path, and system hierarchy was revised by replacing recursions in depth-first order with iterations over the enumerated objects in both directions. The reuse of already calculated

sums at deeper levels of the hierarchy was preserved by making sure that child nodes received a higher index than their parents did.

Figure 4 compares the iterative aggregation algorithms to their recursive counterparts on a Linux workstation, again using the SMG2000 data sets as an example. As can be seen, the new algorithm significantly reduced the aggregation time in the majority of the cases. The fact that the calculation of the metric tree was slower in two cases can be ignored because this tree is calculated only once during the entire display session, whereas the call tree and the system tree have to be frequently re-calculated. In the end, the previously often slow response times that were noticeable especially at large scales could be substantially reduced.

6 Conclusion

In this paper, we have presented a number of measures to improve the scalability of the CUBE display. Although none of the changes required truly novel algorithms, they nonetheless led to tangible results in the form of a much quicker generation of input data sets, a simplified usage on remote desktops, significantly reduced memory requirements, and noticeably improved interactive response times. While certainly adding to the convenience of the overall user experience, the most important accomplishment, however, is that CUBE and the tools depending on it can now be used to study the runtime behavior of applications running at substantially larger scales.

References

1. F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore, *An Algebra for Cross-Experiment Performance Analysis*, in: Proc. of the International Conference on Parallel Processing (ICPP), pp. 63–72, IEEE Society, Montreal, Canada, August 2004.
2. Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr, *Scalable Parallel Trace-Based Performance Analysis*, in: Proc. 13th European PVM/MPI Users' Group Meeting, vol. 4192 of *LNCS*, pp. 303–312, Springer, Bonn, Germany, September 2006.
3. Sameer Shende and Allen D. Malony, *The TAU Parallel Performance System*, International Journal of High Performance Computing Applications, **20**, no. 2, 287–331, 2006.
4. B. Krammer, M. S. Müller, and M. M. Resch, *Runtime Checking of MPI Applications with MARMOT*, in: Proc. of Parallel Computing (ParCo), pp. 893–900, Málaga, Spain, September 2005.
5. Accelerated Strategic Computing Initiative, "The ASC SMG2000 benchmark code", `http://www.llnl.gov/asc/purple/benchmarks/limited/smg/`, 2001.
6. H. Brunst and W. E. Nagel, *Scalable Performance Analysis of Parallel Systems: Concepts and Experiences*, in: Proc. of the Parallel Computing Conference (ParCo), pp. 737–744, Dresden, Germany, 2003.