Scaling Score-P to the next level*

Daniel Lorenz[1] and Christian Feld[2]

[1] Laboratory for Parallel Programming,
Technische Universität Darmstadt,
Darmstadt, Germany
lorenz@cs.tu-darmstadt.de
[2] Jülich Supercomputing Centre,
Forschungszentrum Jülich GmbH,
Jülich, Germany
c.feld@fz-juelich.de

Abstract
As part of performance measurements with Score-P, a description of the system and the execution locations is recorded into the performance measurement reports. For large-scale measurements using a million or more processes, the global system description can consume all the available memory. While the information stored process-locally during measurement is small, the memory requirement becomes a bottleneck in the process of constructing a global representation of the whole system. To address this problem we implemented a new system description in Score-P that exploits regular structures of the system, and results, on homogeneous systems, in a system description of constant size. Furthermore, we present a parallel algorithm to create a global view from the process-local information. The scalable system description comes at the price that it is no longer possible to assign individual names to each system element, but only enumerate elements of the same type. We have successfully tested the new approach on the full JUQUEEN system with up to nearly two million processes.

Keywords: Performance analysis, data compression, exascale computing

1 Introduction

Contemporary high performance systems increase their performance mostly through increased parallelism. However, to effectively utilize the available parallelism, applications need to use the provided resources efficiently. Performance analysis tools help to find scalability bottlenecks and, thus, are essential for increasing the scalability of applications. To measure the performance, the performance measurement system has to scale to the same level as the application does.

*This material is based upon work supported by the US Department of Energy under Grant No. DE-SC0015524 and by the German Federal Ministry for Education and Research (BMBF) under Grant No. 01IH13001.

Score-P [8] is a performance measurement tool for parallel applications. One of its design goals was high scalability. It instruments an application and records performance data for each execution location. To store its measurement data, it shares the memory with the application. The user can specify the amount of memory that Score-P may use during measurement and, thus, can control how much memory is available for the application and the performance measurement.

Beside performance data, Score-P records also the so called *definitions*, meta-data that describes the program structure, system hierarchy and metrics. The definitions are kept in memory, because they define the data realm. However, on systems with millions of cores, the description of the system hierarchy and the execution locations can require more memory than is available for a single process. For the system definition, Score-P creates a tree structure in which the root node represents the whole system and the rest of the nodes represent elements of the system hierarchy (e.g., node, rack) and execution locations (process, thread) (see Figure 2). In the beginning of a measurement, every process creates the local system definitions, which are the definitions for the nodes on the path from the root node to the nodes that represent itself. In a finalization step the local information is gathered to build the global system tree which is a representation of the whole system. The global system tree is required to correctly collate the local data from each process into a global performance measurement report and for later analysis of the data.

Creating the local system definition records requires only a few nodes and, thus, little memory. The memory bottleneck occurs when Score-P tries to build and write the global system tree. The size of the system tree grows linearly with the size of the system, especially with the number of processes and threads. On large systems, the system tree description can consume multiple hundreds of megabytes of memory. On JUQUEEN, the system used for our tests, only 256 MB of memory are available per hardware thread. This amount must be shared between the operating system, the application, and the measurement system. Figure 1 shows the memory footprint during the Score-P finalization of an instrumented “hello world” program. The memory footprint is dominated by the system tree definitions. If we run with 64 processes per node on JUQUEEN, we are not able to run the measurement with 262,144 processes because the size of the system tree description would be too large to fit into the available memory.

In this paper, we present a system tree description that is based on the PERI-XML profiling format proposal for data exchange [11]. It exploits regular structures to reduce the size of the system description. On homogeneous systems, the size of the system tree description has constant size and is independent of the scale on which the application runs. In addition, we present a distributed algorithm to create the new global system tree description. We implemented our approach in Score-P and tested it at full-scale on the JUQUEEN system. The memory requirements of this algorithm depend on the per-process-memory needed to implement MPI communicators. For large scale systems efficient MPI communicator implementations are crucial. Previous work from Moody et al. [12] and Langer et al. [9] describes methods to implement MPI communicators with logarithmic size or even constant size requirements per process.

The paper is organized as follows. Section 2 contains a survey of related work. Section 3 describes the system tree definitions. Section 4 describes the algorithm. Section 5 describes the evaluation of our method. We close with a conclusion in Section 6.

2 Related Work

Our system description is based on the PERI-XML proposal for a profiling data exchange format [11]. Like our approach, PERI-XML takes advantage of the regularity of large systems.

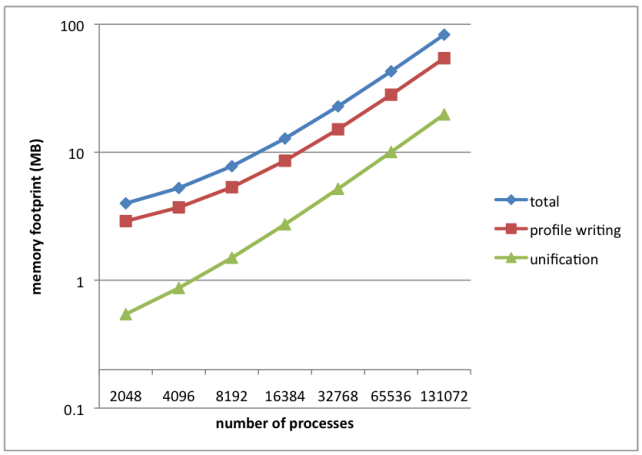

Figure 1: The memory footprint of the Score-P finalization step for an instrumented "hello world" program.

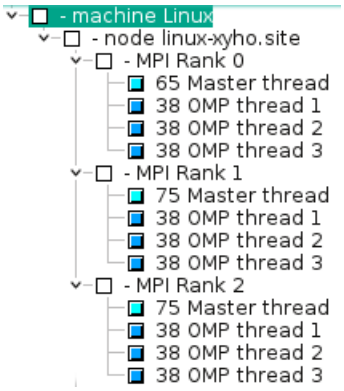

Figure 2: The Cube display of the system tree.

However, it provides only a specification of the format without any implementation that creates descriptions from local information. The format was never widely adopted by HPC tools. ParaProf [3] can read data in PERI-XML format, but, to the best of our knowledge, there are no tools that can write PERI-XML files.

The amount of data at large-scale is a challenge to every performance analysis tool. The amount of trace data in particular can become huge; in the advent of exascale applications and systems the amount of profile data becomes a challenge, too. One approach to reduce the amount of data is to use standard data compression methods, such as leading-zero compression and zlib compression. They can be applied to traces [16] or profiles, e.g., Cube [15] uses zlib compression.

In addition to general techniques, trace-specific methods exist as well. Knüpfer et al. developed a trace compression algorithm [7] that searches for matching event sequences along the time axis and compresses them. ScalaTrace [14] uses intra- and inter-node compression to reduce the size of MPI event traces. The authors also exploit repetitions of event sequences along the time axis to reduce the overall trace size. Another way to reduce the size of MPI traces is to extract the loop nesting structure during measurement from its event flow graph [1] and using sophisticated compression mechanisms [2] to further reduce the amount of data recorded.

Another approach that reduces the size of the trace data is stratified sampling [5]. It adjusts the sampling frequency to the measured application, and exploits equivalence classes of processes to reduce the number of processes to be sampled. However, all of these methods target the measurement data and do not consider excessive system descriptions.

In [10], the authors aggregate the data of multiple threads within a process with different strategies. This results in an implicit reduction of the number of execution locations and a shrinking of the system description. However, the aggregation takes place only within a process. The system description still grows linearly with the number of processes.

Another data reduction approach is filtering [17]. Filtering reduces the size of the call tree by removing certain nodes. In most cases, the user can specify regions that are included in the measurement with white lists or black lists. Mußler et al. [13] presented criteria for automatic filters based on static analysis of the executable.

Our algorithm depends on MPI-communicators with low memory requirements. The implementation of MPI communicators on large scale systems face the challenge of limited memory, too. More space-efficient MPI communicator implementations are possible [12, 9]. Similarly, the representation of communicators within the Score-P measurement system need to be memory-efficient for large-scale systems. Geimer et al. [6] describe a representation of MPI communicators in Score-P, which requires only constant memory per process.

3 System Tree Definitions

The Score-P [8] measurement system captures some meta-data to describe the measured performance data. This meta-data is called *definitions*, e.g., the system tree description is called *system tree definitions*. The current system definitions in Score-P are structured as a tree, where every node represents a system unit, e.g., a rack, a node, a node card, a midplane, a CPU, a core or the whole system. Every child node represents a system unit that is a part of the system unit represented by its parent node and can contain smaller units that are represented by its child nodes, e.g., the root node represents the whole machine. The machine may consist of several compute nodes that are children of the machine node. A node may have multiple CPUs, which are represented by its children. Furthermore, Score-P adds execution streams, e.g., processes or threads, to the system tree and, thus, associates them with an execution location. In the implementation, Score-P creates one data record for every system tree node. In the following we call this type of system tree definitions *single-node definitions*. The system definitions are kept in memory during measurement and are written to disk either in the CUBE4 format [15] (profiling mode) or in OTF2 format [4] (tracing mode). Figure 2 shows a system tree as displayed in the Cube GUI, which allows to browse CUBE4 profiles. The size of the system tree grows linearly with the number of nodes in the system tree. Considering that the system tree also contains processes and threads, the memory consumption of the system tree definitions may become excessively large.

However, most large systems have a regular, homogeneous structure, e.g., each rack has the same number of nodes. Even so called inhomogeneous systems usually have a significant amount of regularity, e.g., half of the nodes have accelerators, whereas the other half has no accelerators. We can exploit the homogeneity by designing a system tree definitions format that describes the patterns that can be observed in the system tree definitions. Therefore, we aggregate sub-trees of the system tree that have the same structure, keeping one representative, and store a value which specifies how many copies of this sub-tree exist. This aggregation can be performed on each level of the system tree. We call this type of definitions *sequence definitions*, because every data record defines a sequence of system tree elements of the same type. Figure 3 shows an example for the sequence definitions.

The sequence definition records are based on the PERI-XML proposal [11]. The new system tree sequence definitions have (i) a class field that describes the system unit type, (ii) A field that contains the number of copies of this element and (iii) a list of child definitions to describe the structure of each copy. If a system is not completely homogeneous it may have multiple entries in the list of children at any level. Figure 4 show the example of an inhomogeneous system. Thus, these definitions are flexible enough to describe also completely irregular systems. However, inhomogeneity increases the size of the definitions. If the system contains no regularity, the sequence definitions degenerate to the full system tree with one data record per system tree node. However, we do not expect large-scale systems to be without any regularity. In fully homogeneous systems, the system tree definition depends only on the depth of the tree and not on the number of elements and, thus, have constant size.

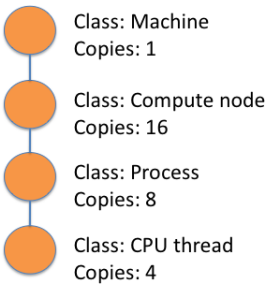

Figure 3: This is an example of sequence definitions for a system tree. The system consists of one machine with 16 compute nodes. There are 8 processes on each compute node and each of the processes has 4 threads.

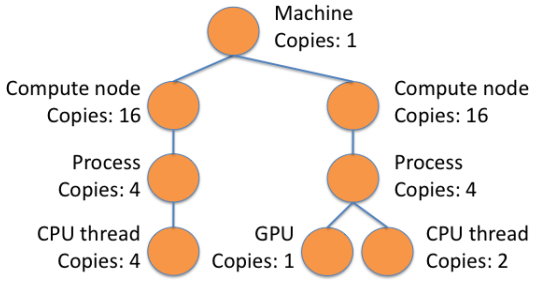

Figure 4: This is an example of the sequence definitions for an inhomogeneous system. This machine has two types of compute nodes: 16 nodes belong to the first type, which runs 4 processes with 4 threads per process; the other 16 nodes belong to the second type, which has only 2 threads per process and a GPU context in each of the processes.

With sequence definitions we can describe the system architecture. However, we also want to associate an execution location or any other system tree node with recorded performance data. Thus, we need a mechanism to identify individual system tree nodes. Therefore, we simulate a depth-first traversal of the whole system tree by iteratively traversing a system tree sequence definition as many times as the number stored in the *copies* field. This traversal defines an enumeration of the nodes in the system tree that can provide an unique index to reference individual system tree nodes.

4 Distributed System Tree Creation

The memory requirements of the sequence definitions may be small, but we do not gain any scalability if the algorithm that creates the sequence definitions has a memory footprint that grows linearly with the system size, even if just on one process. In this section we describe a distributed algorithm that extracts the regular patterns and, on homogeneous systems, has a memory footprint that depends on the size of the MPI communicators, the depth of the system tree and the number of execution locations per process. With constant-size MPI communicators, the algorithm has also a memory footprint that is independent of the number of processes on homogeneous systems.

On every process, the Score-P measurement system obtains the system location where it executes, the system tree node for this process and its ancestor nodes. Furthermore, it creates the nodes for the threads and other execution locations that belong to the process. Thus, we have a tree on every process that branches only on the execution location level. The definitions are still single-node definitions that we will convert into sequence definitions. Figure 5 shows the definitions that are obtained on every process.

The algorithm processes the system tree level by level from the leaves to the root. First, it creates sequence definitions for the process and its execution locations from the already obtained single-node definitions. For every type of execution location it counts the number of occurrences and creates a single sequence definition record for this type. In addition it creates a sequence definition for the process itself. At this first step there are still just process-local definitions, as

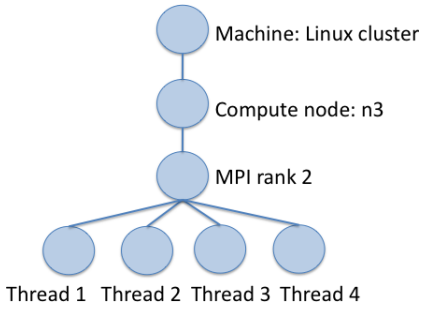

Figure 5: The single-node definitions that are obtained by Score-P at measurement initialization on every process. It includes the definitions for the process, its execution locations, and the system tree nodes on the path to the root.

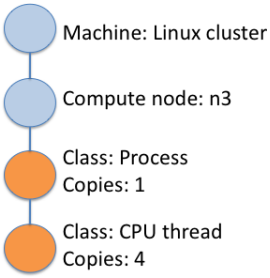

Figure 6: This figure continues the example from Figure 5. It shows the process-local system tree definitions after the conversion of process definitions and execution location definitions to sequence definitions. Blue circles mark single-node definitions. Orange nodes mark sequence definitions.

we have not yet aggregated definitions among processes. Figure 6 continues the example from Figure 5 and shows the partial sequence definitions for a process and its execution locations.

As a second step we need to create the sequence definitions for the remaining levels of the system tree. Therefore we need to identify the children of each node at the current depth level d. To do so, every system tree node is assigned to a process that executes the computation for this node. The goal of this step is to create a MPI communicator $C_{node}(N)$ for every node N in the current level d; this communicator comprises all the processes that computed the sequence definitions for the child nodes of N (see Figure 7).

We are going to create the new communicators by splitting existing ones by color, where the color is obtained from hashing unique node names, e.g., the host names. When running into hash-collisions, we need to reiterate the splitting on the inconclusive communicator. We are following [12], where the authors present an algorithm to split communicators that works on every user defined data as color. Furthermore, the authors propose to store communicators as a chain where every process stores only its neighbors. As long as only a split is required they found that their hash-based communicator split requires only $O(1)$ memory and $O(logP)$ time, where P is the number of processes.

Within each $C_{node}(N)$, we aggregate the definitions into rank 0 of $C_{node}(N)$. Whenever a process merges the definitions from another process p, it compares whether the sequence definitions sub-tree from p matches one of its own. If it does, the number of copies is increased by one. Otherwise, the sequence definitions from p are added as an additional child to the current definition. To compute $C_{node}(N)$ we need an MPI communicator $C_{depth}(d)$ that contains all the processes that computed the sequence definitions for nodes in level $d+1$. At the beginning of the algorithm, every process constructed its own sequence definitions. Thus, the initial communicator $C_{depth}(D)$ is `MPI_COMM_WORLD`. For the computation on the next level $d-1$, we also need to create $C_{depth}(d-1)$, which consists of all the processes that are rank 0 in one of the node specific communicators of the current level. Figure 8 shows the continuation of the example. Afterwards, the aggregation continues at the next higher level, until we finally have processed the whole system.

To aggregate the data within $C_{node}(N)$, we construct a binary spanning tree over all processes, with rank 0 in $C_{node}(N)$ as the root node. Every process receives the sequence definitions

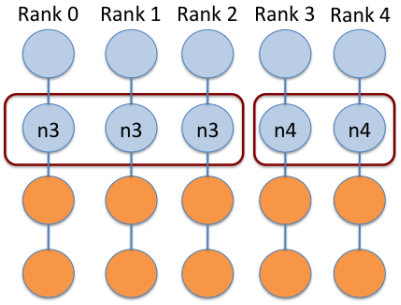

Figure 7: This figure continues the example from Figure 6. To convert the next level of single-node definitions to sequence definitions, we need to identify the processes that belong to the same system tree node. The figure shows multiple processes and the system tree definitions that are available on these processes. The red line encircles matching nodes that we want to aggregate to a sequence definition.

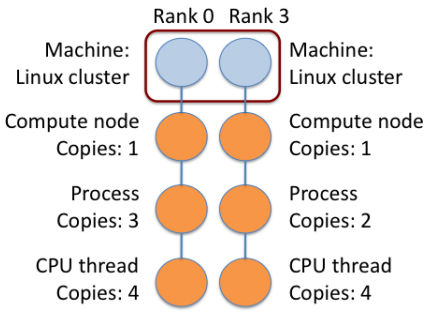

Figure 8: This figure continues the example from Figure 7. Rank 0 and Rank 3 have aggregated all definitions from the other processes that belonged to the same compute node. Furthermore, they converted the single-node definition of the compute node into a corresponding sequence definition. Thus, these two ranks belong to the communicator $C_{depth}(0)$. On the top, the rank number of the process in `MPI_COMM_WORLD` is shown.

from its two children in the spanning tree, merges them with its own, and sends the merged data to its parent. In addition it needs to store mapping data for its children.

To tell each process its place in the global definitions, we need to calculate its index and distribute it backwards on the same path we used for aggregating the definitions. Starting at the global root with its index 0 and using the mapping information every process has collected, we are able to calculate and distribute back the indices to all processes.

5 Evaluation

To evaluate the sequence definition algorithm, we compared the heap memory footprint and the execution time of the Score-P measurement finalization using either the established single-node definition algorithm or the proposed sequence definition implementation. As the generation of the system tree and its writing to file is independent of the application's behavior, we used a simple "hello world" program that we instrumented with Score-P for the evaluation. Using such a program with minimal intrinsic resource consumption facilitates the comparison of the two algorithms.

We performed our measurements on JUQUEEN, an IBM Blue Gene/Q system with 28 racks, 28,672 nodes and 458,752 cores. Each rack consists of two midplanes. Each midplane has 16 node-boards and each node-board has 32 compute nodes. An application can run with up to 64 ranks per compute node. On the whole machine, we can run applications with up to 1,835,008 processes. Every node has 16 GB memory. Thus, with 64 ranks per node, every rank has 256 MB memory that needs to be shared between executable, application data, MPI and measurement data. Hybrid applications that use MPI in combination with threads get more memory per rank which relaxes the memory restrictions. Running with the maximum number of ranks per node is the hard case for our measurement system. We repeated the time

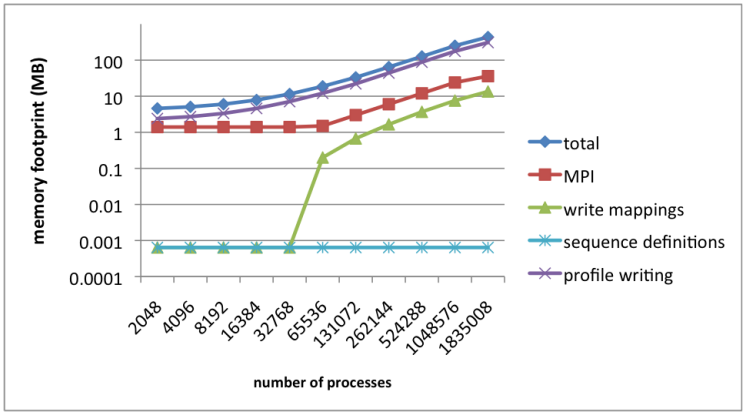

Figure 9: The heap memory footprint during Score-P finalization with sequence definitions. "MPI" shows the memory allocated for MPI communicators during the calculation of the sequence definitions. The memory footprint shown for "sequence definitions" contains all memory allocations during the calculation of the sequence definitions, except MPI. The "write mappings" shows the buffer allocations for collecting the mappings. "Profile writing" shows the heap memory allocations during writing the profile to disk.

measurements ten times. The memory footprint is deterministic. For the execution time we show the median.

With sequence definitions, we were able to run measurements of the "hello world" code on the whole machine with 1,835,008 MPI ranks. Figure 9 shows the memory footprint during the Score-P finalization. We measured the memory footprint for MPI communicators separately from the memory footprint of the sequence definition creation. The sequence definitions alone used 672 bytes of memory; this value was constant for all numbers of processes. This matches the theoretical analysis in Section 4. The MPI communicators consumed constant space as long as the whole job ran on a single midplane; running on a larger partition caused the memory requirements to grow linearly with the number of processes. An ideal MPI implementation could reduce the MPI memory footprint to a constant value, though.

The main contributor to the total memory footprint is the Cube profile writing. The Cube library writes data items as a sequence of values for all execution locations. Thus, we need to gather the data items from all locations which requires a buffer that grows with the number of locations. Furthermore, we need to create a new communicator that contains all ranks in the order of their index in the depth-first system tree traversal of the sequence definitions. A parallel profile writing mechanism could limit the required buffer sizes for data collection to a constant amount and a write-through approach in Cube could reduce the memory allocations inside of the library. However, this is subject to future work. To write the mappings, we use the same technique as for gathering the profile data. We observed buffer allocations only when using more than one midplane, see the jump in the graph for writing the mappings. This is similar to the observations of the MPI communicators. For 131,072 ranks, the total heap memory footprint for sequence definitions is a factor of 2.5 smaller than for single-node definitions.

Figure 10 shows the comparison of the execution time of the Score-P finalization with sequence definitions and Score-P finalization with single-node definitions. With 131,077 processes, the profile writing was 91 times faster for sequence definitions than for single-node definitions. This factor grows with the number of processes, because the sequence definitions have constant

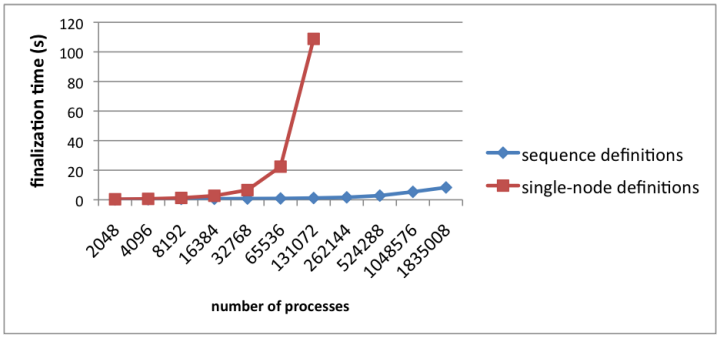

Figure 10: The execution time of the Score-P finalization with sequence definitions versus with single-node definitions.

size while the single-node definitions grow linearly with the number of processes.

6 Conclusion

The size of the system description can limit the scalability of a performance measurement and analysis tool. However, most large scale systems have a mostly regular structure. Even so called heterogeneous systems have a limited number of different component types. Sequence definitions, a system description that exploits the regularity of the system, use only constant memory as the number of processes increases. They can be constructed with a constant memory footprint if MPI communicators have constant size. In this case the memory requirements for a system description using sequence definitions depend on the depth of the system tree and are independent of the number of processes. The presented algorithm relies on an efficient implementation of MPI communicators. However, on large scale systems we expect efficient MPI implementations.

The new algorithm solved a scalability limitation in Score-P which was caused by excessive memory consumption of the previous algorithm that used single-node definitions. We were able to prove that with sequence definitions, Score-P scales to nearly two million ranks using the whole JUQUEEN system. We could achieve this even though the memory constraints were very tight and the memory requirements per process of the MPI communicator implementation is not constant. The price we have to pay for sequence definitions is that system tree elements cannot have individual names any longer, but are constructed from a type and a running number. However, mappings can circumvent this limitation to some extent.

References

[1] X. Aguilar, K. Fürlinger, and E. Laure. Automatic on-line detection of MPI application structure with event flow graphs. In *Proceedings of the 21th International Euro-Par Conference on Parallel Processing (Euro-Par '15)*, Vienna, Austria, Aug. 2015.

[2] X. Aguilar, K. Fürlinger, and E. Laure. Online MPI trace compression using event flow graphs and wavelets. In *Proceedings of the 2016 International Conference on Computational Science, ICCS*, San Diego, USA, June 2016.

[3] R. Bell, A. D. Malony, and S. Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *Euro-Par 2003 Parallel Processing: 9th International*

Euro-Par Conference Klagenfurt, Austria, August 26-29, 2003 Proceedings, pages 17–26, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[4] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf. Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In *Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium, August 30 – September 2 2011*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2012.

[5] T. Gamblin, R. Fowler, and D. A. Reed. Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*, pages 1–12, Apr. 2008.

[6] M. Geimer, M.-A. Hermanns, C. Siebert, F. Wolf, and B. J. N. Wylie. Scaling performance tool MPI communicator management. In *Proc. of the 18th European MPI Users' Group Meeting (EuroMPI), Santorini, Greece*, volume 6960 of *Lecture Notes in Computer Science*, pages 178–187. Springer, Sept. 2011.

[7] A. Knüpfer and W. E. Nagel. Compressible memory data structures for event-based trace analysis. *Future Generation Computer Systems*, 22(3):359–368, 2006.

[8] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of 5th Parallel Tools Workshop, 2011, Dresden, Germany*, pages 79–91. Springer Berlin Heidelberg, Sept. 2012.

[9] A. Langer, R. Venkataraman, and L. Kale. Scalable algorithms for constructing balanced spanning trees on system-ranked process groups. In *Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, pages 224–234, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[10] D. Lorenz, S. Shudler, and F. Wolf. Preventing the explosion of exascale profile data with smart thread-level aggregation. In *Proc. of ESPT2015: Workshop on Extreme Scale Programming Tools, held in conjunction with the Supercomputing Conference (SC15), Austin, TX, USA*, pages 1:1–1:10. ACM, Nov. 2015.

[11] B. Mohr, M. Schulz, D. Gunter, K. Huck, W. Spear, and X. Wu. A proposal for a profiling data exchange format. In *CScADS Performance Tools Workshop*, 2009.

[12] A. Moody, D. H. Ahn, and B. R. de Supinski. Exascale algorithms for generalized MPI_Comm_Split. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 9–18, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] J. Mußler, D. Lorenz, and F. Wolf. Reducing the overhead of direct application instrumentation using prior static analysis. In *Proc. of the 17th Euro-Par Conference, Bordeaux, France*, volume 6852 of *Lecture Notes in Computer Science*, pages 65–76. Springer, Sept. 2011.

[14] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696 – 710, 2009.

[15] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr. Cube v4: From performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343–1352, June 2015.

[16] M. Wagner, A. Knüpfer, and W. E. Nagel. Enhanced encoding techniques for the Open Trace Format 2. *Procedia Computer Science*, 9:1979–1987, 2012. Proceedings of the International Conference on Computational Science, (ICCS) 2012.

[17] M. Wagner and W. Nagel. Strategies for real-time event reduction. In *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *Lecture Notes in Computer Science*, pages 429–438. Springer Berlin Heidelberg, 2013.