Preventing the explosion of exascale profile data with smart thread-level aggregation

Daniel Lorenz
Laboratory for Parallel Programming
Technische Universität Darmstadt
Darmstadt, Germany
lorenz@cs.tu-darmstadt.de

Sergei Shudler
Laboratory for Parallel Programming
Technische Universität Darmstadt
Darmstadt, Germany
shudler@cs.tu-darmstadt.de

Felix Wolf
Laboratory for Parallel Programming
Technische Universität Darmstadt
Darmstadt, Germany
wolf@cs.tu-darmstadt.de

ABSTRACT

State of the art performance analysis tools, such as Score-P, record performance profiles on a per-thread basis. However, for exascale systems the number of threads is expected to be in the order of a billion threads, and this would result in extremely large performance profiles. In most cases the user almost never inspects the individual per-thread data. In this paper, we propose to aggregate per-thread performance data in each process to reduce its amount to a reasonable size. Our goal is to aggregate the threads such that the thread-level performance issues are still visible and analyzable. Therefore, we implemented four aggregation strategies in Score-P: (i) SUM – aggregates all threads of a process into a process profile; (ii) SET – calculates statistical key data as well as the sum; (iii) KEY – identifies three threads (i.e., *key threads*) of particular interest for performance analysis and aggregates the rest of the threads; (iv) CALLTREE – clusters threads that have the same call-tree structure. For each one of these strategies we evaluate the compression ratio and how they maintain thread-level performance behavior information. The aggregation does not incur any additional performance overhead at application run-time.

General Terms

Algorithms, experientation, measurement

Keywords

Performance analysis, data compression, exascale computing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ESPT2015 November 15-20 2015, Austin, TX, USA
Copyright is held by the owner/authors. Publication rights licensed to ACM.
Copyright 2015 ACM 978-1-4503-3997-1/15/11 ...$15.00.
DOI: `http://dx.doi.org/10.1145/2832106.2832107`.

1. INTRODUCTION

Score-P [10] is a performance measurement tool that targets massively parallel codes. The Score-P measurement system is used by Scalasca [5], Vampir [8], Periscope [1] and TAU [18] to record performance data. It instruments applications and records performance data either as a stream of events in a trace file or as a call tree profile which is stored as a CUBE4 report [4]. The Score-P measurement system records data for each thread separately to avoid additional synchronization for accesses to common data structures during the profile generation. In general, the required memory per thread for the profile data is small compared to the memory available for each thread. However, the total size of the profile data grows linearly with the number of threads in an application.

The estimated number of threads in future exascale systems is in the order of a billion threads. This means that if a performance analysis tool records just one datum with a length of 8 bytes, e.g. one timestamp, for each thread, then the total amount of data would be 8 GB. The growing size of the profile data is one of the challenges for performance analysis tools which should be able to scale to an exascale system. However, not only the total number of threads is expected to grow, but also the number of threads per process is expected to be in the order of 100 to 1000 threads per process. Thus, we propose to aggregate the per-thread data within each process, thereby, significantly reducing the amount of data. Aggregating all the per-thread data might reduce the user's ability to investigate thread-level performance behavior. However, the user rarely inspects all of the individual threads, and if a deeper insight into the performance behavior of the thread-level parallelism is required, he would most likely inspect only a few candidates. Thus, we want to keep enough data to be able to catch significant performance issues in an application.

In this paper, we present four different aggregation strategies. Each strategy aggregates the threads in a different way, but still keeps track of the total number of aggregated threads.

- SUM: Provides only aggregated per-process data. In most cases, it means summing up thread-specific values.
- SET: In addition to the sum, it calculates some key statistical values. In particular, it calculates the max-

imum value among all thread, the minimum, and the sum of squares, which allows to calculate the standard deviation.

- KEY: Keeps the data for the initial thread, the fastest thread, and the slowest thread. All other threads are aggregated. This strategy keeps the data for the slowest thread, because it assumes that in order to improve the overall performance the user has to improve the performance of the slowest thread. The strategy also keeps the fastest thread since it would most probably have a different behavior than the slowest one. This would allow comparing the slowest thread to a candidate with a different behavior. Since the master thread often plays a special role in some applications, the KEY strategy keeps it too. Furthermore, we can calculate a mean over the aggregated threads that can also be used for comparison. The KEY strategy assumes that most often single members of the aggregated threads are not investigated in detail.

- CALLTREE: Aggregates threads that have the same call tree structure. It is based on the assumption that different behavior is often reflected by a difference in the call tree. It implicitly provides information about the number of different call trees that are within the application and allows to analyze the differences between threads that have different call trees.

The rest of the paper is organized as follows. We start with a survey of related work in Section 2. Afterwards, Section 3 presents an overview about the CUBE data format and a common workflow of a CUBE profile inspection. This provides the foundation for Section 4, which describes the aggregation strategies and their implementation in more detail. In Section 5 we evaluate the compression ratio of these strategies, and in Section 6 we apply the aggregation strategies to codes which show performance issues with the thread-level parallelism. Section 7 concludes the paper and outlines some ideas for future work.

2. RELATED WORK

The growing size of performance data was already addressed in a number of other studies. The CUBE4 profile format [4], which we use in our work, uses the standard zlib compression mechanism to reduce the file size. Furthermore, if a metric contains many zero values, CUBE can reduce the space consumption by using a different internal data representation compared to when metrics have mostly non-zero values. These compression techniques are used in addition to our aggregation strategies.

A similar approach to thread aggregation, especially to the CALLTREE method, is the clustering mechanism for time series profiles [19]. Time series profiles generate separate sub-profiles for each visit to a source code region. Typically, each iteration of the main loop creates its own sub-profile represented by a sub-tree in the call tree. If there are many iterations in the code, time series profiles can become very large. Thus, similar iterations are clustered to reduce the profile size. In contrast to our approach, however, the call-path dimension is clustered.

PerfExplorer [6] is a tool for data mining on parallel profiles. It can apply hierarchical and k-means clustering on profiles. The main purpose of the clustering is data analysis.

Since trace data is usually much larger than profile data, compression of trace data became already an issue on smaller scales. Standard compression methods, such as leading-zero compression and zlib compression, can be applied to traces too, as described in [20]. In addition to the general compression techniques, trace-specific methods exist as well. Knüpfer et al. developed a trace compression algorithm [9] that searches for matching event sequences along the time axis and compresses them. ScalaTrace [16] 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 approach that reduces the size of the trace data is stratified sampling [3]. It adjusts the sampling frequency to the measured application, and exploits equivalence classes of processes to reduce the number of processes to be sampled.

Filtering of frequently called functions can significantly reduce the size of a trace. However, it also removes the information about filtered events from the measurement data. Profile sizes are not significantly reduced by filtering, but it is used for measurement overhead reduction. Filtering requires the user to define appropriate filter files before profiling the application. Lorenz et el. [15] presented criteria for automatic filters based on static analysis of the executable. TAU [18] contains an option, called throttling, to stop recording events after a defined amount of occurrences. More details about filtering and throttling are given in [21].

When debugging large-scale applications, inspecting all MPI tasks may become a burden comparable to inspecting all the threads. Laguna et al. [11] use hierarchical clustering to determine outliers that exhibit uncommon behavior and are of special interest for inspection.

3. ANALYSIS WORKFLOW WITH CUBE

This paper presents a number of aggregation strategies that reduce the detail-level of performance data. However, we want to aggregate the data in such way that it still maintains the necessary information to identify, classify and solve performance bottlenecks. First, we want to describe the CUBE data model and the most common workflow for the evaluation of a CUBE profile (Figure 1). The goal of this workflow is to find criteria for relevant information.

Even at a smaller scale than exascale, performance analysis measurements usually produce a large amount of data. In order to understand and evaluate the data, the user starts with an overview and digs down into the source of the problem. Thus, to actually find performance bottlenecks, without requiring the user to manually sift through multiple GB of data, the performance analysis tool must first present a high-level indication of a certain problem and then guide the user to the place where he should dig deeper to find the cause of the bottleneck.

The CUBE data model has the three dimensions: metric, call path and system. The elements in each dimensions can be arranged in a tree structure. Let M be the number of metrics and C the number of call paths. In the system dimension, CUBE stores data only for an execution location, e.g, a CPU thread. The execution locations are always leaves in the system tree; thus, the set of all execution locations S defines the possible data indices of the system dimension. The data $D(s)$ of each execution location $s \in S$ is a two-

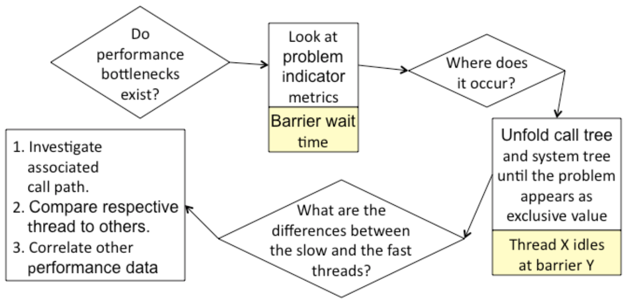

Do performance bottlenecks exist?
Look at problem indicator metrics
Barrier wait time
Where does it occur?
Unfold call tree and system tree until the problem appears as exclusive value
Thread X idles at barrier Y
What are the differences between the slow and the fast threads?
1. Investigate associated call path.
2. Compare respective thread to others.
3. Correlate other performance data

Figure 1: CUBE analysis workflow.

dimensional matrix where the rows represent the calltree dimension and the columns represent the metric dimension:

$$D(s) = \begin{pmatrix} d_{1,1} & ... & d_{1,M} \\ ... & & ... \\ d_{C,1} & ... & d_{C,M} \end{pmatrix}.$$

Any aggregation operation $\oplus \in \{+, *, max, min\}$ that is performed on the data of two threads $s_1, s_2 \in S$ is applied element-wise. Thus, for the data of the two threads

$$D(s_1) = \begin{pmatrix} a_{1,1} & ... & a_{1,M} \\ ... & & ... \\ a_{C,1} & ... & a_{C,M} \end{pmatrix}$$

and

$$D(s_2) = \begin{pmatrix} b_{1,1} & ... & b_{1,M} \\ ... & & ... \\ b_{C,1} & ... & b_{C,M} \end{pmatrix}$$

the aggregation result of the data of the two threads is:

$$D(s_1) \oplus D(s_2) = \begin{pmatrix} a_{1,1} \oplus b_{1,1} & ... & a_{1,M} \oplus b_{1,M} \\ ... & & ... \\ a_{C,1} \oplus b_{C,1} & ... & a_{C,M} \oplus b_{C,M} \end{pmatrix}.$$

Processes are represented by parent nodes of the execution locations. With P we define a division of S where each $p \in P$ contains the subset of execution locations that belong to the same process. Thus, p represents a process.

The CUBE graphical user interface presents each dimension in a separate panel, with all the three panels located next to each other (Figure 2). In the default configuration, the left panel shows the metric tree, the middle panel the call tree, and the right panel the system tree. However, the order of the panels can be changed. The left panel shows values that are aggregated over the other two dimensions. The middle panel shows the value that is selected in the left panel, aggregated over the dimension of the right panel, and the right panel shows the values that are selected in the middle and left panel. In the default configuration, the left panel shows the metric values aggregated over all call paths and the whole system tree. The middle panel shows the values of the selected metric for the call tree dimension aggregated over the system dimension. The right panel shows the value of the selected metric and the selected call path on each system tree node.

The tree structure allows the user to interactively explore each dimension by expanding or collapsing tree nodes. If a node is collapsed, it does not show its child nodes and the value next to it is *inclusive*. Inclusive value means that it includes the values of all the child nodes, e.g., the execution time of this region including all regions called by this region. On the other hand, if a node is expanded, the child nodes are shown in the panel right below it and the value next to it is *exclusive*. Exclusive value means it does not include the child nodes values. By default, the viewer starts with collapsed trees, i.e., all dimensions show only collapsed root nodes.

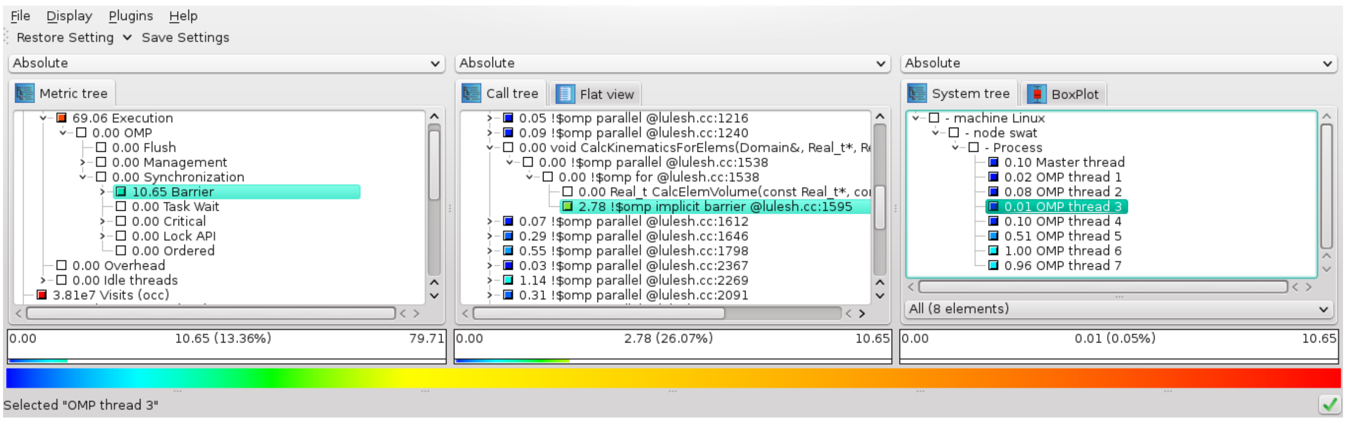
In the first step of exploring a CUBE report, the user needs to know which performance problems exist in his application, if at all. Problem indicator metrics, such as time spent in MPI wait states or OpenMP synchronization constructs, provide an immediate indication of the significance of the relevant performance bottleneck. At this point, the overview values are aggregated over all call paths and all execution locations. If a performance bottleneck exists, it raises the question of "Where?". The user first selects the indicator metric he wants to investigate, and then expands the call tree and the system tree until the indicator metric value is visible as exclusive value. Expanding the call tree dimension allows the user to identify the source code region and the calling context information. Furthermore, expanding the system tree dimension allows the user to identify the relevant execution locations.

In some cases, pointing to the place where the performance bottleneck occurred may already be sufficient to solve it. Otherwise, further investigation is needed. This typically involves various techniques, such as comparing the suspected execution location to other locations and correlating other metrics (e.g., hardware counters) with the execution time and the indicator metric. For example, let us assume that OpenMP synchronization time points us to a barrier where all threads, except the master, wait a significant amount of time. It tells us that we have an imbalance and the master thread needs much longer time than all worker threads. We may now want to investigate why it takes so long and look at the code that is executed before that barrier. One possibility might be that there is a function that takes more time on the master thread than on the other threads. At this point, we might want to continue correlating other metrics to find the reason for this difference.

4. AGGREGATION STRATEGIES

The Score-P measurement system records a separate profile for each thread. In general, the profile of each thread consumes only a small fraction of its memory, and it means that during the measurement we can still record per-thread data. However, the collective data amount over all the threads can become too large. It is the system dimension that poses the challenge, and to tackle it we want to aggregate the performance data of multiple threads. Aggregating the data of the threads within a process affects only the information that is related to the thread-level parallelism. It maintains all performance-relevant data on the process level and higher. Because estimations for exascale systems predict some 100 to 1000 threads per process, this already provides significant data reduction potential. Furthermore, each process can perform the aggregation without any additional interprocess communication, since all data is available in a shared scope. The aggregation happens in the measurement finalization step, avoiding additional overhead at measurement time.

We developed four aggregation strategies, each representing a different approach, to find a compromise between compression and information reduction.



```
Absolute                                   Absolute                                               Absolute

69.06 Execution                            0.05 !$omp parallel @lulesh.cc:1216                    - machine Linux
  0.00 OMP                                 0.09 !$omp parallel @lulesh.cc:1240                      - node swat
    0.00 Flush                             0.00 void CalcKinematicsForElems(Domain&, Real_t*, Re      - Process
    0.00 Management                          0.00 !$omp parallel @lulesh.cc:1538                        0.10 Master thread
    0.00 Synchronization                       0.00 !$omp for @lulesh.cc:1538                           0.02 OMP thread 1
      10.65 Barrier                              0.00 Real_t CalcElemVolume(const Real_t*, co           0.08 OMP thread 2
      0.00 Task Wait                             2.78 !$omp implicit barrier @lulesh.cc:1595            0.01 OMP thread 3
      0.00 Critical                        0.07 !$omp parallel @lulesh.cc:1612                          0.10 OMP thread 4
      0.00 Lock API                        0.29 !$omp parallel @lulesh.cc:1646                          0.51 OMP thread 5
      0.00 Ordered                         0.55 !$omp parallel @lulesh.cc:1798                          1.00 OMP thread 6
  0.00 Overhead                            0.03 !$omp parallel @lulesh.cc:2367                          0.96 OMP thread 7
  0.00 Idle threads                        1.14 !$omp parallel @lulesh.cc:2269
3.81e7 Visits (occ)                        0.31 !$omp parallel @lulesh.cc:2091                    All (8 elements)

0.00      10.65 (13.36%)      79.71        0.00      2.78 (26.07%)      10.65                     0.00      0.01 (0.05%)      10.65

Selected "OMP thread 3"
```

Figure 2: The unaggregated profile after selecting the indicator metric and expanding the call tree and the system tree to determine where the synchronization time occurs.

4.1 SUM

In the first strategy, called SUM, the threads of a process p are replaced by an artificial execution location s_a, which contains the aggregated values of all threads. The aggregation is an element-wise sum of the data of all threads:

$$D(s_a) = \sum_{s \in p} D(s)$$

This aggregation strategy has the best compression ratio. The size of the system dimension shrinks to the number of processes.

4.2 SET

The goal of the second strategy, called SET, is to provide the user with information about the variation of data through statistical key values such as minimum, maximum and standard deviation. Thus, in addition to the sum over all threads, it also calculates the maximum value, the minimum value, and the sum of the squares over all threads. Moreover, each tuple contains the number of samples which equals to the number of aggregated threads in p that have at least one visit to the corresponding call path. In combination with the number of samples, the sum of squares allows to calculate the standard deviation. In principle, it would be possible to extend this set with further statistical data, such as median and quartiles. Thus, the aggregated process data can be represented as $p' = \{s_{sum}, s_{min}, s_{max}, s_n, s_{sum2}\}$, where s_{sum} stores the sum, s_{min} the minimum, s_{max} the maximum, s_n the number of samples, and s_{sum2} the sum of squares.

For the calculation of $s_{sum}, s_{min}, s_{max}, s_{sum2}$, we use element-wise operations:

$$\begin{aligned} D(s_{sum}) &= \sum_{s \in p} D(s) \\ D(s_{min}) &= min_{s \in p} D(s) \\ D(s_{max}) &= max_{s \in p} D(s) \\ D(s_{sum2}) &= \sum_{s \in p} D(s)^2 \end{aligned}$$

To calculate the number of elements, this strategy calculates first a $M \times C$ matrix $N(s)$ for every thread s, which contains the value 1 on all elements in rows that represent a call path that was visited at least once by s and 0 otherwise.

$$D(s_n) = \sum_{s \in p} N(s)$$

One drawback in this strategy is that when it is used with the CUBE GUI it does not allow, for all the members of the set, to convert from inclusive to exclusive values and vice versa. For example, it is not possible to calculate the minimum exclusive time from the inclusive values.

4.3 KEY

The third aggregation strategy, called KEY, is based on the idea that for the analysis of the thread-level parallelism not all threads are necessary. Sometimes only a few *key threads* are of particular interest. If we want to increase the overall performance of the process, we need to improve the performance of the slowest thread first. Everything else would, most probably, not improve the runtime. Therefore, we are mostly interested in the data of the slowest thread s_{slow}. On the other hand, we want to be able to compare this data to a thread which is different from s_{slow}. For that purpose, we are also interested in the other extreme, namely the fastest thread s_{fast}. Since also the initial thread s_0 plays a distinct role in many applications, we decided to record its data as well. All the remaining threads are aggregated, and the average in this case may also be a good comparison candidate, especially if comparing to the fastest thread does not provide sufficient insight.

To determine which thread is the fastest or slowest thread, we classify the call path of the application based on region type information that is provided by the instrumentation. For example, the OPARI2 [12, 14] instrumentation of OpenMP regions and the OpenMP tools interface [2] proposal provide the means to mark a barrier as such a region. We distinguish regions that are doing work from regions that indicate synchronization, idling, or waiting. We calculate the execution time that a thread spent inside those regions that we classified as doing work. The thread with the highest execution time in work regions is the slowest thread. The thread with the smallest execution time in work regions is the fastest thread. The initial thread is excluded from the determination of the fastest and slowest thread, because it is recorded anyway.

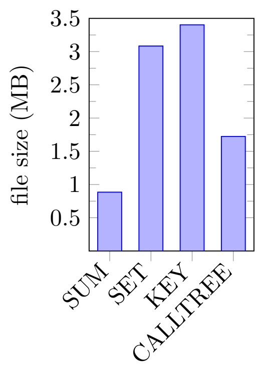


Figure 3: The size of the CUBE profile in MB for the generated test profile with 100 call paths, 7 metrics and 128 processes. 90% of the call paths are parallel call paths. The file size of the aggregated profile data is independent of the number of threads per process.

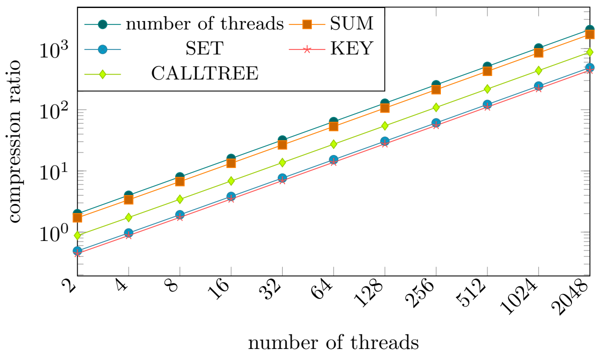


Figure 4: The compression ratio for all aggregation strategies in dependence on the number of threads per process.

As a result, we substitute the locations in p with the four elements $p' = \{s_0, s_{slow}, s_{fast}, s_a\}$, where s_a contains the aggregated threads. For the aggregation we perform an element-wise calculation of all elements.

$D(s_a) = \sum_{s \in p \setminus \{s_0, s_{fast}, s_{slow}\}} D(s)$

Our selection of threads covers the most cases where thread-level parallelism is evaluated. It also provides some additional analysis hints by pointing the user to the most problematic and most extreme threads.

4.4 CALLTREE

The fourth aggregation strategy, called CALLTREE, aggregates threads that have the same call tree structure. The call tree of two threads $s_1, s_2 \in S$ match if for all call path $c \in C$ the equivalence

$visits(s_1, c) > 0 \iff visits(s_2, c) > 0$

is valid. Hereby, $visits(s_i, c)$ denotes the number of visits to call path c by thread s_i. It results in a non-predetermined number of aggregated clusters of threads. Clustering in itself is already an additional automatic analysis, because it provides an overview of how many groups of threads with distinct call tree exist. It is based on the assumption that differences in the call tree structure may indicate a different behavior. For the aggregation of threads within a cluster, we use again the element-wise calculation.

5. COMPRESSION

One of the motivating goals for the aggregation was a reduced profile size. To determine the compression ratio, we generated profiles for every aggregation strategy and one without any aggregation. The compression ratio is the size of the profile without aggregation divided by the aggregated profile size. The generated profiles contain 7 metrics, 100 call paths, and had 128 processes. Figure 3 presents the file sizes, and Figure 4 shows the compression ratio against the number of threads per process.

The amount of data per process is constant for all four aggregation strategies. Thus, the size of the aggregated profile file was independent of the amount of threads for all aggregation strategies. The compression ratio grows with the number of threads in the application, because the uncompressed original becomes larger. The SUM strategy has the best compression ratio. However, the compression ratio is a factor of 1.35 to 1.36 smaller than the number of threads because it uses an additional metric, which stores the number of aggregated threads. Furthermore, the profile metadata section requires some additional space to store information about the metric dimension and the call tree dimension, which has constant size.

The compression ratio of the SET strategy is approximately 4.2 times smaller than the number of threads. It needs to store four additional values per metric. The size of the number-of-threads value is only 4 bytes, while the size of all other values is 8 bytes. Thus, the ideal compression ratio would be 4.5 times smaller than the number of threads. We contribute the difference to the constant sized metadata section.

The KEY strategy stores four 8 bytes values per metric/call path pair. The compression ratio is approximately 4.6 times smaller than the number of threads. The ideal compression rate would be 4 times smaller than the number of threads per process. The SET strategy has a slightly better compression ratio than the KEY strategy although the ideal compression ratio is worse. The SET strategy, however, is more likely to store the same number multiple times, which is exploited by CUBE's inbuilt zlib compression.

The CALLTREE strategy creates 2 clusters for this application. Thus, it needs to store 2 values per original metric/call path pair. It uses a similar approach to the KEY strategy to store the data, and creates an artificial location for each cluster. The compression ratio is 1.97 better than KEY which is explained by the fact that it stores only half the values. The compression ratio is 2.3 times smaller than the number of threads per process. The result of 2 clusters is typical for such OpenMP applications, where one master thread is in one cluster and the worker threads produce all the same call tree and, therefore, form the second cluster.

The compression ratio is mostly determined by the number of threads per process. Other factors that affect the compression ratio in a minor way are the number of metrics and the number of processes. The data section of the CUBE profile grows linearly with the number of metrics. However, the metadata section for the system dimension and the call tree dimension stay constant. Because only the data section shrinks during the aggregation, the compression ratio becomes larger if the number of metrics grow. Similarly, if the number of processes grow, the data-to-metadata ratio grows as well, which leads to a slightly better compression ratio.

6. ANALYSIS OF THREAD-LEVEL PARALLELISM

In Section 3, we identified the following steps which usually occur in the analysis of the profile data:

1. Which types of performance bottlenecks exist? This question is answered by the aggregated value of indicator metrics over the whole measurement. Because the value of interest is aggregated anyway, providing only aggregated values on a process level does not curtail the potential insights in this step.

2. Where does it occur? Unfolding the call tree dimension until the indicator metric value is shown as exclusive value works also with aggregated threads. However, identifying the execution location on thread level is affected by aggregation.

3. For further investigation of a problem cause, the first strategy is to compare a problematic thread to other threads.

4. Another investigation strategy is to correlate metrics.

To evaluate the retained information, we measured multiple applications that contain a specific performance bottleneck with every aggregation strategy. Afterwards we compare the process of finding the performance bottleneck in the aggregated profiles to the same process with unaggregated profile. All tests ran on an Linux cluster node equipped with two 2.67 GHz Intel XEON quad-core processors with two-way simultaneous multi-threading.

6.1 Imbalance

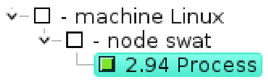
For this test, we injected some imbalance in a parallel region of the Lulesh code [7] by forcing a static schedule with a chunk size of 2000. We executed the code on 8 threads and calculated 27,000 particles. The loop iterates over the particles, which means that 5 threads execute 4000 iterations, the 6th threads gets 3000 iterations and two threads execute 2000 iterations.

In order to obtain the required profiles, we instrumented the artificial test program with Score-P, and then performed one measurement run for every evaluated aggregation strategy and one measurement without any aggregation. The generated profiles contained the following metrics: execution time, number of visits, minimum and maximum execution time at a single visit. To enhance the very basic metric set of the Score-P measurement with further metrics that are able to highlight a particular performance bottleneck, we applied the `cube_remap` tool on the resulting profiles. The `cube_remap` tool classifies every call path and creates additional metrics related to the classification. For example, the execution time gets sub-metrics that allow to divide execution time by user code, or OpenMP regions and further sub-divide time spent in OpenMP regions (e.g, separate synchronization regions). We can use the execution-time-in-synchronization-regions metric to reveal imbalances by identifying places where the application spends a significant time for synchronization.

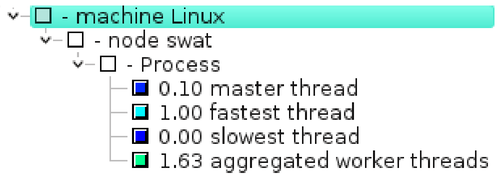
6.1.1 Without aggregation

The CUBE profile shows that 10.65 s (13.37% of the whole execution time of the application) were spent in OpenMP implicit barriers. If we select this metric and expand the call tree, it shows that the implicit barrier at the end of the *parallel for* construct in lulesh.cc at line 1595 produced the largest amount of the synchronization time. Full expansion of the system tree shows that the synchronization time is unevenly distributed. Five of the threads show very little synchronization time of less than 0.1 s at the barrier, while two other threads waited approx. 1 s each and `OMP thread 5` spent 0.51 s at the barrier. Figure 2 shows the CUBE interface after expanding the call tree and the system dimension.

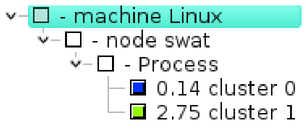
If we correlate the visits and the execution time metric to the synchronization time, we see that the threads that spent little time in the implicit barrier, visited the function `CalcElemVolume` twice as often as the threads that spent 1 s in the barrier. Also, the execution time of the first group of threads is twice the execution time of the waiting threads. The different number of visits stem from a different number of iterations that the threads execute. The reason is that the chunk size in the static schedule clause of the parallel region statement is too large.



(a) SUM



(b) KEY



(c) CALLTREE

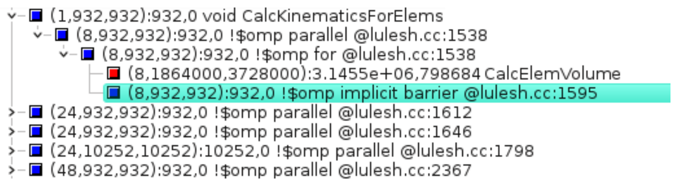
Figure 5: The system dimension of the aggregated profile using the SUM, the KEY and the CALLTREE strategy.

6.1.2 SUM

Figure 5a shows the profile of the SUM aggregation strategy. The indicator metric still shows that 2.94 s of the total time was spent in the implicit barrier. However, we can not tell in which threads the synchronization happened, or whether it is evenly distributed among all threads. The SUM strategy, therefore, allows to identify the performance problem type and the source code location where it occurred. Although this might already provide valuable hints, it is impossible to identify the specific thread or perform further thread-level analysis.

6.1.3 SET

In the SET strategy, the conversion of inclusive values to exclusive values does not work anymore for some members of the data set. Thus, the `cube_remap` tool does not compute the indicator metrics. A first approach to iden-



```
(1,932,932):932,0 void CalcKinematicsForElems
  (8,932,932):932,0 !$omp parallel @lulesh.cc:1538
    (8,932,932):932,0 !$omp for @lulesh.cc:1538
      (8,1864000,3728000):3.1455e+06,798684 CalcElemVolume
      (8,932,932):932,0 !$omp implicit barrier @lulesh.cc:1595
(24,932,932):932,0 !$omp parallel @lulesh.cc:1612
(24,932,932):932,0 !$omp parallel @lulesh.cc:1646
(24,10252,10252):10252,0 !$omp parallel @lulesh.cc:1798
(48,932,932):932,0 !$omp parallel @lulesh.cc:2367
```

Figure 6: The call path dimension of the aggregated profile using the SET strategy. The highlighted metric is the number-of-visits.

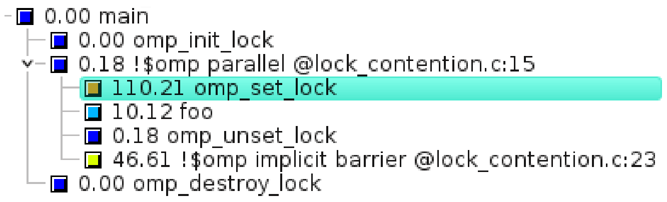
tify performance bottlenecks is to expand the call tree and look for execution time in a suspicious call path. Although scanning the call tree for suspicious call paths requires some extra work, we are still able to find the barrier that has the largest execution time. From the minimum and maximum values we can derive that there is at least one thread that has almost no waiting time while at least one thread must have a waiting time of 1 s. This shows some imbalance, although we do not know which threads suffer from it and wait at the barrier.

For further analysis, we can investigate the visits metric for `CalcElemVolume` (Figure 6). It shows that the average number of visits is 3.15 million with a standard deviation of 0.8 million, the maximum is 3.73 million and the minimum is 1.86 million visits. Most likely, the thread that spent most time at the barrier is the thread that has the least visits to `CalcElemVolume`. However, we can not conclude this for sure, because the statistics for each metric/call path pair is independent. Thus, in general we can not correlate metrics or call path data. For example, the thread that had the minimum or maximum execution time at the implicit barrier does not need to be the same thread that produces the minimum or maximum number of visits to `CalcElemVolume`. Therefore, the distribution may provide some extra information to estimate that a certain performance bottleneck exists, but it does not provide enough information to compare threads and correlate metrics and call path information.

6.1.4 KEY

Figure 5b shows the profile of the KEY aggregation strategy. Again, the synchronization time metric points to the implicit barrier. Expanding the system tree dimension shows that the fastest thread waits at this barrier, while the slowest thread has no waiting time here. This indicates that there is an imbalance. Since the aggregated threads contain some synchronization time at the barrier as well, this indicates that more threads wait for the slowest thread. Although we do not know exactly how much time the other threads wait at the barrier, the slowest thread can serve as an example for the kind of threads where we need to improve the performance.

The KEY strategy allows us to compare the slowest thread to the fastest thread, and to correlate the execution time and number of visits for `CalcElemVolume`. This reveals that the slowest thread has twice the number of visits and twice the workload than the fastest thread. It proves that the number of iterations is unequal and this, subsequently, causes the imbalance. Thus, the KEY strategy provides enough information to identify the causes of the problem.



```
0.00 main
  0.00 omp_init_lock
  0.18 !$omp parallel @lock_contention.c:15
    110.21 omp_set_lock
    10.12 foo
    0.18 omp_unset_lock
    46.61 !$omp implicit barrier @lock_contention.c:23
  0.00 omp_destroy_lock
```

Figure 7: The call tree of the artificial program with lock contention. It shows the execution time metric.

6.1.5 CALLTREE

Figure 5c shows that the CALLTREE strategy distinguishes between the initial thread, which has a visit to the main function, and the worker threads. We can find the implicit barrier by using the synchronization time metric. In our example, the initial thread (cluster 0) spends only 0.14 s at the barrier, while the aggregated worker threads (cluster 1) spend 2.75 s there. Since cluster 1 contains 7 of the 8 threads, it has larger execution times nearly everywhere. Nevertheless, some of the worker threads must have spent much more than 0.14 s at the implicit barrier. We can compare the behavior of the two clusters and also correlate metrics and call path. Thus, we can discover that the number of visits and the execution time in `CalcElemVolume` in cluster 0 is larger than the mean of this value in cluster 1. Because the algorithm decided to distinguish between the initial thread and the worker threads, it suggests that the differences between the initial thread and the worker threads are larger than the differences among the worker threads. The clustering decision is based on call paths outside the parallel region, and though the initial thread has the longest execution time, it may be misleading to interpret that all or most of the worker threads are idle. To make the performance bottleneck more obvious, a distinction between threads that wait at the barrier and threads that execute two chunks of iterations would be better.

Nevertheless, indicating that the initial thread executes more iterations than the average worker thread puts us on the right track to balance the work in the loop more evenly. Similar to the KEY strategy, the total number of time and visits spent in `CalcElemVolume` leads to the conclusion that the number of iterations is not evenly distributed among the threads. Since the clustering decision did not separate the threads according to the execution time, the solution in this case is more obscured than in the KEY strategy. Although the call tree structure is an important factor to determine clusters, there might be better clustering criteria.

6.2 Lock contention

To evaluate the possibility of finding a lock contention with aggregated profile data, we created an artificial test program in which the access to the function `foo` is protected by a lock. Without aggregation, the profile shows that most of the time is spent inside OpenMP synchronization regions. In particular, 110.21 s are spent in the OpenMP Lock API and 46.61 s in the implicit barrier. We select the execution-time-in-OpenMP-Lock-API metric and expand the call tree. This leads us to the lock where the application spends most of its time.

Figure 7 shows the execution time for all the call paths. From the call path, we can derive the source code location and the lock that causes the lock contention. At this point, we have all the necessary information to detect and analyze the lock contention.

We note that the application spent also a significant amount of time at the barrier in the end, and this usually indicates an imbalance. If we compare the time spent in `foo` and the parallel region, we can see that the execution of the user code has no significant variation between the threads, but the time spent in `omp_set_lock` varies significantly. The sum of the time spent in the implicit barrier and `omp_set_lock` is the same for each thread. It means that the imbalance is caused by the lock contention.

As a result, we do not need thread-specific information to identify the causes of the lock contention. The SUM aggregation strategy provides us with sufficient information. As already discussed in Section 6.1, thread-specific information is only needed to conclude that the imbalance is caused by the lock contention.

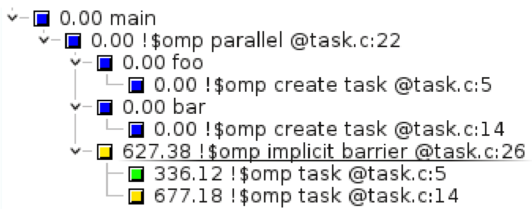
6.3 False sharing

Our third test case is an artificial code that contains false sharing. To indicate false sharing, we would need to associate data accesses to memory locations and cache misses to prior data accesses to another memory location. This information is currently not available in Score-P, and it does not provide an indicator metric for false sharing. Although we can record cache misses, they might be caused by a number of issues and it is hard to tell whether a certain amount of cache misses actually indicates a performance problem.

During the measurements we also recorded level 1 data cache misses, level 1 data cache hits, and level 2 data cache misses. Because of the false sharing, the *parallel for* loop contains 27.7 million level 2 data cache misses. It means that 3.3% of all data accesses inside the *parallel for* loop result in a cache miss. In general, 3.3% cache misses do not necessarily imply a performance bottleneck. However, 27.7 million level 2 data cache misses in a program that has an aggregated computing time of 5 s contribute significantly to the overall runtime. After resolving the false sharing, the same program had an aggregated runtime of only 0.06 s and only 3005 level 2 data cache misses.

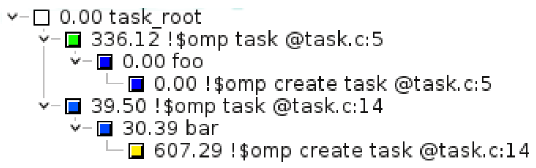
The relevant information that we can extract from the unaggregated profile is the source code location where most of the cache misses occur. We correlate the number of cache hits to estimate the fraction of the cache misses. The system dimension is of interest if false sharing occurs only on a few threads. It allows then to identify the threads where we have a significantly higher number of cache misses. Knowing the involved threads may help to identify the variables that share a cache line.

All aggregation strategies allow to find source code locations where a high number of cache misses occur. The level of support each strategy provides for identifying the threads with higher cache misses is different. The SUM strategy does not provide any support for thread identification. The SET strategy shows whether there is a variation among the threads, but does not allow to identify the threads with a high number of cache misses. The KEY strategy shows whether the slowest thread has a large number of cache misses compared to other threads. Thus, the KEY strategy can highlight one involved thread, but not the threads it interferes with. If false sharing does not occur in the slowest thread, it does not limit the performance. Although it might



```
0.00 main
  0.00 !$omp parallel @task.c:22
    0.00 foo
      0.00 !$omp create task @task.c:5
    0.00 bar
      0.00 !$omp create task @task.c:14
    627.38 !$omp implicit barrier @task.c:26
      336.12 !$omp task @task.c:5
      677.18 !$omp task @task.c:14
```

Figure 8: Implicit tasks call path in the task granularity test program (shown metric is execution time).



```
0.00 task_root
  336.12 !$omp task @task.c:5
    0.00 foo
      0.00 !$omp create task @task.c:5
  39.50 !$omp task @task.c:14
    30.39 bar
      607.29 !$omp create task @task.c:14
```

Figure 9: Explicit tasks call path in the task granularity test program (shown metric is the execution time).

occur in other threads, the user should first concentrate his efforts on fixing the slowest thread. With the CALLTREE strategy the identification of a subset of threads depends on whether the threads that show false sharing differ also in their call tree. If they do, the CALLTREE structure separates them into different clusters, otherwise they end up in the same cluster and, therefore, unidentifiable.

6.4 Task granularity

In our fourth use case, we want to evaluate a task-based parallelization problem. One of the most common performance analysis targets is to identify tasks with inappropriate granularity [17]. For this purpose, we use an artificial program that has two task constructs. One task construct creates tasks that execute for 1s. The other task construct creates tasks that do not perform any work besides the recursive task creation.

Figures 8 and 9 present a CUBE profile with call paths of implicit and explicit tasks. The former shows stub nodes at execution locations along the main call tree, and the latter shows a subtree for every task construct. The Score-P task profiling mechanism and resulting profile data is explained in [13] in more detail.

The main call tree in Figure 8 shows that, basically, the whole time was spent inside the implicit barrier at the end of the parallel region. Inside the barrier, 1013.3s were spent executing the tasks, whereas 627.38s are the exclusive execution time of the barrier. One possible explanation for this time is an overhead caused by task management activities, imbalance, or complex task dependencies, which have insufficient parallelism. Figure 9 shows that the tasks created from the task construct at `task.c:5` spent all the time in the task region doing some work. The average execution time per task was 1s. On the other hand, the tasks created from the task construct at `task.c:14` spent 607s out of 677s creating new tasks. Only 69.9s were spent outside the recursive task creation. This indicates that the overhead is large and the tasks are too small. The average execution time for a task was $14\mu s$, including the measurement overhead.

Since tasking provides a form of automatic work balancing and since the test program contains no task dependencies, the possible imbalances must be smaller than the largest task. Therefore, we can exclude imbalance as a major contributor to the exclusive time in the barrier. On the other hand, we spend 607s creating tasks. Former task analysis examples showed that task switches and task completion can require roughly the same amount of execution time which will appear as exclusive execution time in the barrier [13]. In principle, task dependency structures may limit parallelism. Although it would be possible to investigate these dependencies with trace data, our current profiling approach does not provide the means to do so.

The whole analysis used no thread-level data, but only the call tree dimension. Thus, all of the presented aggregation strategies are able to provide the necessary information. The uncompressed profile, the SET strategy, and the KEY strategy are able to show that the execution time in the barrier is evenly distributed among all threads supporting the claim that there is no imbalance. Attributing performance data of tasks to threads is not trivial, especially for migrating tasks. Furthermore, the task schedule is dynamic and may suffer from run-to-run variation. As a matter of fact, it is one of the goals of the tasking paradigm that the programmer should not have to worry about the distribution of tasks among the threads. In this case, it is reflected by the fact that the per-thread data is not necessary for the analysis of task performance behavior. A separation of tasks for other criteria, e.g., the recursion depth [13], provides much more important information.

7. CONCLUSION AND FUTURE WORK

This paper presents the strategies SUM, SET, KEY, and CALLTREE that aggregate the performance data of threads within a process. Although the aggregation reduces the granularity of the performance information, some strategies still provide sufficient information to identify and investigate thread-level performance bottlenecks. These aggregation strategies differ in compression ratio and the availability of thread-level information.

The SUM strategy has the best compression ratio, but provides the least thread-specific information. Although lack of information prevents thread-level root-cause analysis, it is still possible to detect the presence of thread-level performance issues and to identify the source code locations where they happen. Since this strategy provides a good compression rate, it is a good choice for cases in which the focus of the analysis works sufficiently well with process-level data, such as analysis of MPI, or if the only purpose is to get an overview of the potential bottlenecks.

The amount of data produced by the KEY strategy equals approximately to the amount of data produced when there are 4 to 5 threads per process. It allows to identify performance bottlenecks on the thread level, to compare between threads, and to correlate the metric and call path data. Although this strategy reduces the set of threads to which we can apply these analysis techniques, it still captures the slowest thread information and this is the thread that needs performance improvement in most cases. If multiple bottlenecks exist, the aggregation may limit the in-depth analysis to the most dominant bottleneck. Nevertheless, the KEY strategy provides a good tradeoff between compression ratio and available thread-level information for the analysis of large scale applications.

The SET strategy provides an aggregated value per process and some additional statistical information about the distribution of the performance data among the threads. This information, however, does not allow to compare threads or correlate call paths. Since the KEY strategy provides better analysis possibilities with a very similar compression ratio, the KEY strategy is preferable to the SET strategy.

The CALLTREE strategy is the only strategy that does not provide a predicable compression ratio that depends on the number of threads per process, the number of metrics, and the number of processes. The compression ratio, in this case, depends also on the call tree structure. CALLTREE provides information to investigate a performance bottleneck using thread comparison and metric/call path correlation if the different behavior of the thread is reflected by a difference in the call tree. In our example, using the call tree structure as the only clustering criterion provided some hints, but did not result in clusters that would have been best suited for performance bottleneck analysis. Nevertheless, clustering of threads is a promising approach that provides some useful information, such as how many different thread groups can be distinguished. However, developing a meaningful distance function for performance data is not trivial and subject to future research – especially if it is to be used for analyzing any kind of performance issue and not just one specific bottleneck.

The effects of heterogeneous architectures with accelerators and coprocessors are not covered by this paper and subject to further research. One approach might be to separate each type of execution location and then apply data reduction on each of the subgroups. Probably the call trees of CPU threads, accelerator code and other execution location types differ. Thus, a combination of the CALLPATH strategy to separate the execution location types and any other strategy within each execution location group might provide reasonable results.

Besides the conclusions for the particular compression strategies, one future guiding conclusion is that the amount of data becomes too large when we store the full, fine granularity, per-thread information. Thus, our more promising approaches perform various degrees of automated analysis on the full thread-level information to determine which parts of the information should be stored with fine granularity and which parts of the information can be accumulated to a coarser granularity. Even if it as simple as the KEY strategy, which determines simply the slowest and fastest thread, the automated analysis approaches can reduce the amount of data while retaining its usefulness. These approaches form a wide field that provides numerous opportunities for further research, and the simple approaches in this paper are a baseline for more elaborate automated analysis and granularity selection strategies. Advanced analysis methods may, for example, include clustering algorithms based on Score-P's region type classification, execution time, communication patterns or any other metric. Another promising approach subject to future research would be to cluster processes, record all threads of one process from every cluster and aggregate the threads on the other processes.

8. ACKNOWLEDGEMENT

This material is based upon work supported by the US Department of Energy under Grant No. DE-FG02-13ER26158 / DE-SC0010668 and by the German Federal Ministry for Education and Research (BMBF) under Grant No. 01|H13001.

9. REFERENCES

[1] S. Benedict, V. Petkov, and M. Gerndt. PERISCOPE: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer, Berlin/Heidelberg, 2010.

[2] A. E. Eichenberger, J. M. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. OMPT: OpenMP tools application programming interfaces for performance analysis. In *Proc. of the 9th International Workshop on OpenMP (IWOMP), Canberra, Australia*, number 8122 in LNCS, pages 171–185, Berlin / Heidelberg, 2013. Springer.

[3] 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.

[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 *Proc. of the Conference on Parallel Computing (ParCo), Aachen/Jülich, Germany*, pages 645–652, September 2007. *Minisymposium Scalability and Usability of HPC Programming Tools.*

[5] M. Geimer, F. Wolf, B. J. N. Wylie, D. B. Erika Abraham, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, Apr. 2010.

[6] K. Huck and A. D. Malony. PerfExplorer: A performance data mining framework for large-scale parallel computing. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 41–52, Nov. 2005.

[7] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[8] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir performance analysis tool-set. In *Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, pages 139–155, Stuttgart, Germany, July 2008. Springer-Verlag.

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

[10] 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.

[11] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, and B. Rountree. Large scale debugging of parallel tasks with automaded. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 50:1–50:10, New York, NY, USA, 2011. ACM.

[12] D. Lorenz, R. Dietrich, R. Tschüter, and F. Wolf. A comparison between OPARI2 and the OpenMP tools interface in the context of Score-P. In *Proc. of the 10th International Workshop on OpenMP (IWOMP), Salvador, Brazil, September 2014*, volume 8766 of *LNCS*, pages 161–172. Springer International Publishing, Sept. 2014.

[13] D. Lorenz, P. Philippen, D. Schmidl, and F. Wolf. Profiling of OpenMP tasks with Score-P. In *Proc. of the 41st International Conference on Parallel Processing Workshops (ICPPW), Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, pages 444–453, Sept. 2012.

[14] B. Mohr, A. D. Malony, S. 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.

[15] 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.

[16] 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.

[17] D. Schmidl, P. Philippen, D. Lorenz, C. Rössel, M. Geimer, D. an Mey, B. Mohr, and F. Wolf. Performance analysis techniques for task-based OpenMP applications. In *Proc. of the 8th International Workshop on OpenMP (IWOMP), Rome, Italy*, volume 7312 of *Lecture Notes in Computer Science*, pages 196–209, Berlin / Heidelberg, June 2012. Springer.

[18] S. S. Shende and A. D. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.

[19] Z. Szebenyi. *Capturing Parallel Performance Dynamics*. PhD thesis, RWTH Aachen University, volume 12 of IAS Series, Forschungszentrum Jülich, 2012. ISBN 978-3-89336-798-6.

[20] 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.

[21] 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.