Space-Efficient Time-Series Call-Path Profiling of Parallel Applications

Zoltán Szebenyi
Jülich Supercomputing Centre
52425 Jülich, Germany
RWTH Aachen University
52056 Aachen, Germany
z.szebenyi@fz-juelich.de

Felix Wolf
Jülich Supercomputing Centre
52425 Jülich, Germany
RWTH Aachen University
52056 Aachen, Germany
f.wolf@fz-juelich.de

Brian J. N. Wylie
Jülich Supercomputing Centre
52425 Jülich, Germany
b.wylie@fz-juelich.de

ABSTRACT

The performance behavior of parallel simulations often changes considerably as the simulation progresses — with potentially process-dependent variations of temporal patterns. While call-path profiling is an established method of linking a performance problem to the context in which it occurs, call paths reveal only little information about the temporal evolution of performance phenomena. However, generating call-path profiles separately for thousands of iterations may exceed available buffer space — especially when the call tree is large and more than one metric is collected. In this paper, we present a runtime approach for the semantic compression of call-path profiles based on incremental clustering of a series of single-iteration profiles that scales in terms of the number of iterations without sacrificing important performance details. Our approach offers low runtime overhead by using only a condensed version of the profile data when calculating distances and accounts for process-dependent variations by making all clustering decisions locally.

1. INTRODUCTION

As numerical simulations model the temporal evolution of a system, their progress occurs via a series of discrete points in time. According to this iterative nature, the core of such an application is typically a loop that advances the simulated time step by step — the entire loop often preceded by initialization and concluded by finalization procedures. However, the performance behavior may vary between individual iterations, for example, due to periodically re-occurring extra activities [1] or when the state of the computation adjusts to new conditions in so-called adaptive codes [2]. As we have shown in a recent performance study of the SPEC MPI2007 benchmark suite [3], temporal variations are wide-spread and may appear in highly diverse patterns ranging from gradual changes and sudden transitions of the base-line behavior to both periodically and irregularly occurring extrema. To complicate matters further, the temporal behavior can be subject to significant process-dependent variations, that is, the performance behavior can be a function of both time and space. However, recognizing the relationship between temporal and spatial patterns can be crucial for the understanding of performance problems. For example, the *PEPC* Coulomb solver was found to suffer from a gradually increasing communication imbalance caused by a small group of processes with time-dependent constituency [4]. Sometimes, behavioral deviations may also be caused by outside influences, such as operating system jitter or application interference, which may limit their reproducibility.

Call-path profiling [5, 6, 7, 8, 9], which aggregates performance metrics across the entire execution broken down by call path, is a widely-used method of linking a performance problem to the context in which it occurs. Especially when investigating the use of library functions, call-path information can be critical in tracking performance problems back to their roots in the user code. For example, in the analysis of MPI programs, call-path information is often essential to decide where in the program a communication bottleneck occurs. However, the context expressed in the call path usually reveals little about the temporal evolution of a performance phenomenon. As a result, important insights as to how and why certain behaviors develop may be lost.

On the other hand, writing a separate call-path profile for every iteration of the timestep loop, a special use case of a technique commonly referred to as *phase profiling* [8], may pose scalability problems in terms of the number of iterations that can be captured and analyzed. The most severe restriction arises from the buffer space required to store such a *time-series call-path profile* measurement because adding a time dimension multiplies the amount of data by the number of timesteps. If flushing of profile data buffers is to be avoided during measurement, due to its disruptive impact on the behavior to be observed, time-series call-path profiles can exceed available buffer space — especially when the call tree is large and more than one metric is collected. Moreover, even if the perturbation caused by flushing can be tolerated or compensated for, the aggregate size of the profile across a potentially large number of iterations and processes may hinder post-processing and interactive end-user analysis, which often occurs on a front-end node or

SC09 November 14-20, 2009, Portland, Oregon, USA
(c) 2009 ACM 978-1-60558-744-8/09/11... $10.00

desktop with moderate processing and memory capacity.

In this paper, we present a runtime approach for the (lossy) semantic compression of time-series call-path profiles. Our approach of incremental on-line clustering of single-iteration profiles enables the collection of multi-metric call-path-level data for thousands of iterations, while consuming only a few times the space of a single-iteration profile to circumvent the need to intermittently flush the data to disk. Exploiting repetition in the time-dependent behavior of the target application, we achieve good compression rates without sacrificing important performance details or introducing major artefacts. Since calculating the similarity between two iterations based on one or more metrics and a potentially large number of call paths may be prohibitively time consuming, we keep the runtime overhead low by calculating the separation distance based on only a condensed version of the profile data. At the same time, this measure also improves the quality of the clustering algorithm by reducing the dimensionality of the distance operator. Moreover, to account for the fact that different processes may exhibit different temporal patterns, the clustering is a purely local operation, resulting in a custom-tailored process-local partitioning of the iteration space, not requiring any communication or synchronization at runtime.

The article is structured as follows: We start with a review of related work in Section 2. Then, we provide a brief description of our general profiling methodology together with an outline of the key data structures in Section 3, before explaining our compression algorithm along with our design choices in Section 4. In Section 5, we offer a quantitative evaluation of our approach based on the SPEC MPI2007 benchmark suite in terms of (i) the accuracy of the compressed data, and (ii) the runtime overhead incurred. A detailed large-scale application example with the *PEPC* code presented in Section 6 demonstrates the value of our solution for the performance analysis of long-running applications with significant time-dependent behavior. Finally, in Section 7, we conclude the paper and discuss future work.

2. RELATED WORK

Our approach builds on the concept of *phase-based* performance characterization. The execution of a program can be naturally divided into phases, which form the building blocks of its performance behavior. Since phases have different execution characteristics and may react differently to external stimuli such as changes in the execution configuration or the input problem, it seems reasonable to analyze their performance behavior independently instead of looking at the execution only as a whole. While phases provide a general concept to represent arbitrary logical and runtime aspects of the computation, our approach concentrates only on major timestep loop iterations to allow comparisons between execution intervals that occur on the same logical level.

Phase profiling in the performance system TAU allows the user to obtain a separate profile for every marked program interval, distinguishing between static and dynamic phases [8]. Whereas performance metrics are aggregated across all executions of a static phase, a separate profile object is created for every instance of a dynamic phase. According to this distinction, the time intervals we consider can be classified as instances of a dynamic phase. Similar in spirit, incremental profiling in the OpenMP profiler ompP takes a separate profile for every fixed-sized execution time interval, using elapsed wall-clock time instead of the dynamic program structure as the delimiter between phases [10].

The principle of dividing the execution of a program into intervals and grouping them into phases according to their performance characteristics has already been studied by Sherwood et al. in the context of microprocessor hardware simulation [11]. Using clustering algorithms, they identified representative subsections of the instruction stream of a program that can be used as input for simulations that would otherwise be too slow if fed with the complete stream. The results of these shorter simulations can subsequently be extrapolated to reflect the execution of the entire program. In the context of large-scale parallel applications, various statistical and data mining techniques have been employed to reduce the size and improve the understanding of performance data. While projection pursuit [12] and clustering [13] have been applied to improve the understanding of real-time performance data, Ahn et al. have shown that multivariate statistical techniques provide a useful means of identifying correlations among different hardware performance metrics and of highlighting clusters of similar behavior in process topologies [14]. Moreover, PerfExplorer [15] structures profile data of parallel programs post-mortem by performing hierarchical and k-means clustering on vectors of metric values whose elements correspond to program regions.

Space limitations of highly-structured performance data sets that include a time dimension are most apparent for event traces containing timestamped records for a huge number of program actions. Complete call graphs are able to compress event traces by exploiting repetitive patterns [16], but require their prior existence at full length, as runtime overhead prevents the method from being applied on-line. Similarly, automatic structure extraction starts from very large trace files, explores their internal structure using signal processing techniques, and selects meaningful parts [17]. Whereas the previous approaches target the time dimension, Gamblin et al. lower the overall trace volume by tracing only a subset of the processes [18], with sample size periodically readjusted based on summary performance data sent to a central client process at runtime. Finally, application signatures provide a way of summarizing the time-dependent behavior expressed in historical trace data in a much more compact representation [19]. Here, the temporal evolution of a metric vector is described as a metric trajectory using curve fitting as a compression mechanism, simplifying the comparison between two signatures.

3. PROFILING METHODOLOGY

While our general approach is independent of a particular parallel programming model, we chose single-threaded MPI applications as a starting point, which finds its expression in our selection of test cases and profiling metrics. In the remainder of the paper, we therefore always use the term *process* to denote an independent locus of execution, which may be translated to *thread* in a multithreaded context. Our compression algorithm is intended to be used in the runtime measurement system of Scalasca [20], an open-source toolset for diagnosing inefficiencies in parallel applications written in C, C++ and Fortran on a wide range of HPC platforms. Since in addition to pure MPI, Scalasca

also supports measurement and analysis of OpenMP and hybrid OpenMP+MPI codes, we plan to evaluate our algorithm with respect to those programming models in the near future. Users of Scalasca can choose between two types of performance data: (i) call-path profiles summarized during measurement, and (ii) call-path profiles derived from patterns in event traces. Whereas the former are needed to identify the most resource-intensive call paths in the program, the latter can be used to identify call paths that exhibit a significant fraction of idle time. The result of such a trace analysis is therefore similar in structure to a runtime call-path profile but enriched with higher-level communication and synchronization inefficiency metrics. While in principle our algorithm applies to those data sets as well, we will consider only runtime call-path profiling for now.

3.1 Call-Path Profiling

When configured for runtime profiling, Scalasca aggregates performance metrics such as execution time, message statistics, and hardware counters individually for every thread and call path encountered during the entire program execution.

The metrics collected during measurement of MPI applications are listed in Table 1 with indentation representing the hierarchical relationship between them. They are divided into a subset $M = \{M_1, \ldots, M_m\}$ that is directly measured and another subset $D = \{D_1, \ldots, D_d\}$ derived by later analysis (set in italic). Major measurement overhead and MPI initialization and finalization are typically one-off expenses outside the primary computational loop. Optional hardware counter metrics and specific MPI metrics related to file I/O, collective communication, etc., may not exist or be entirely zero-valued in certain measurements.

Different from sampling-based approaches, all measurements are obtained via the direct instrumentation of function or region entries and exits, either via PMPI library interposition, compiler-generated instrumentation, or source-code preprocessing. During program finalization, Scalasca collates all process-local measurement data sets into a global profile report and writes it to disk.

Scalasca defines call paths as lists of visited regions (usually starting from the *main* function) and maintained in a tree data structure. Thus, a new call path is specified as an extension of a previously defined call path to the new terminal region. When a region is entered from the current call path, any child call path and its siblings are checked to determine whether they match the new call path, and if not a new call path is created and appropriately linked (to both parent and last sibling). Exiting a region is then straightforward as the new call path is the current call path's parent call path. When execution is complete, a full set of locally-executed call paths are defined, which are merged into a global set during program finalization, serving as a basis for later call-tree visualizations.

3.2 Time-Series Profiling

To measure time-dependent application behavior, Scalasca was equipped with phase instrumentation capabilities based on TAU's dynamic phase model [8], which implies that performance data are stored separately for every phase instance. With such instrumentation it becomes possible to distinguish the performance data produced by different iterations after enclosing the timestep loop body with enter and exit markers. Translated to a call-path profile, this means that every iteration populates the call tree with its own set of metric values.

Category	Metric
Time	Total
	Measurement overhead
	Execution
	MPI
	MPI init/exit
	MPI synchronization
	MPI collective synchronization
	MPI communication
	MPI point-to-point communication
	MPI collective communication
	MPI file I/O
	MPI collective file I/O
Counts	Call path visits
	Hardware counters (optional)
	MPI synchronizations
	MPI point-to-point synchronizations
	MPI point-to-point send synchronizations
	MPI point-to-point receive synchronizations
	MPI collective synchronizations
	MPI communications
	MPI point-to-point communications
	MPI point-to-point send communications
	MPI point-to-point receive communications
	MPI collective communications
	MPI collective communications as source
	MPI collective communications as dest'n
	MPI collective exchange communications
	MPI bytes transfered
	MPI point-to-point bytes transfered
	MPI point-to-point bytes sent
	MPI point-to-point bytes received
	MPI collective bytes transfered
	MPI collective bytes incoming
	MPI collective bytes outgoing
	MPI File operations
	MPI File individual operations
	MPI File individual reads
	MPI File individual writes
	MPI File collective operations
	MPI File collective reads
	MPI File collective writes

Table 1: Summary metrics collected and derived by Scalasca.

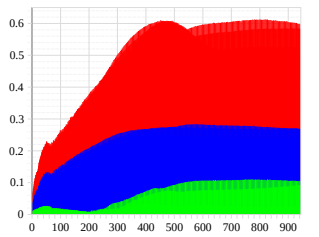
Let C be the set of call paths reached from within the timestep loop body (those outside can be ignored because they are irrelevant to draw comparisons between iterations), I the set of loop iterations, P the set of application processes, and M the directly measured metrics among the ones listed in Table 1. Then we can define a *time-series call-path profile* as a mapping

$$s : (c, i, p) \mapsto \vec{m}$$

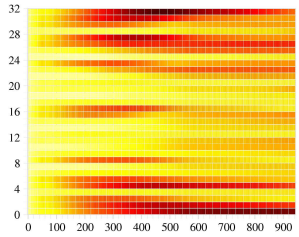
which maps a call path $c \in C$, an iteration $i \in I$, and a process $p \in P$ onto a vector of metric values $\vec{m} \in M_1 \times \ldots \times M_m$ (e.g., the time spent by process p in call path c while executing iteration i). Note that the derived metric values are implied in this definition. The process dimension P can be omitted when considering the data of only a single process, which will often be the case as our compression is a purely process-local operation.

3.3 Analysis

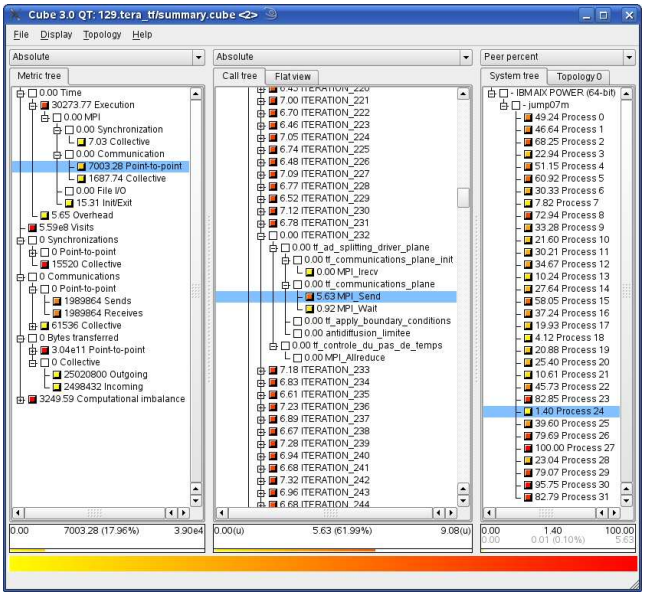
In Scalasca, a time-series call-path profile can be analyzed in several ways. The most common methods are listed below, and illustrated in Figure 1 with one of the SPEC MPI2007 benchmark applications, *129.tera_tf*.



(a) MPI point-to-point communication time iteration graph with maximum (red), median (blue), and minimum (green) values.



(b) MPI point-to-point communication time value map with the red end of the color spectrum indicating higher values.



(c) Scalasca analysis report explorer with MPI point-to-point communication time metric selected (left pane). With 6.55 seconds, iteration 232 of 942 (middle pane) is one of the more expensive ones, and the marked call path to the MPI_Send operations is distinguished by a particularly pronounced imbalance across the 32 processes, with times varying from only 0.01s for rank 24 to 0.41s for rank 27 (right pane).

Figure 1: Different ways of analyzing a 32-process *129.tera_tf* experiment with Scalasca.

- 2-dimensional iteration graphs, which for each iteration (x-axis) show the maximum, median, and minimum values of a given metric (y-axis) calculated from all processes (Fig. 1(a)).
- Value maps, which display the value of a given metric color-encoded as a rectangular block in a 2-dimensional grid with the x-axis representing iterations and the y-axis representing processes (Fig. 1(b)).
- A multi-dimensional tree browser [21] that allows a user to interactively explore the entire performance space spanned by mapping s including the derived metrics and, in particular, to examine the performance behavior of individual call paths separately for each iteration (Fig. 1(c)).

The quantitative evaluation of our algorithm presented in Section 5 assesses the influence of the compression on the expressiveness of these three analysis methods.

4. COMPRESSION ALGORITHM

Our compression algorithm is based on the idea of *incremental clustering*. Clustering is the process of grouping a set of physical or abstract objects into classes of similar objects. A cluster is a collection of data objects that are similar to others within the same cluster and dissimilar to the objects in other clusters [22]. In our case, the objects to be clustered are the metric profiles collected for individual iterations as they are generated while the target application progresses. As the goal is to achieve the best overall characterization, clustering is performed independently for each iteration without considering sequence order. To accurately cover process-dependent variations of temporal patterns, each application process makes independent clustering decisions, which has the additional advantage that neither communication nor synchronization among different processes are required at runtime. Thus, we can restrict our discussion to the actions performed by a single process. The algorithm is lossy in the sense that the compressed data no longer contains all the information necessary to restore the original data, although we will show that the result of such a reverse-transformation comes very close to the original data.

4.1 Clustering

With every new iteration i, a new process-local *iteration profile* $s_i : c \mapsto \vec{m}$ is created, which maps a call path onto a vector of directly measured metric values and which can be stored as a matrix $S^i_{c,m}$. The clustering occurs incrementally because every new iteration initially constitutes a new cluster. Once a predefined maximum number of clusters has been exceeded, the two clusters closest to each other are merged. For each cluster, we store only the iteration indices assigned to it and the mean profile, which is obtained by performing an element-wise arithmetic mean operation on the constituent matrices S^i, exploiting the associativity of the mean operator when merging clusters that represent more than one iteration. Using the disjoint-set forest data structure [23] makes cluster creation and merge plus retrieval of the constituent index set an asymptotically constant-time operation as opposed to a naive linear-time implementation.

4.2 Distance Function

Which clusters are closest to each other is determined based on the distance between cluster means. However, calculating a distance function based on entire profile matrices is inefficient and may also adversely affect the clustering quality, which is sensitive to the dimensionality of the distance function ($|C| * |M|$ if the full matrix is considered). If too large, similarities between objects may be hidden. For this reason, the distance is computed based on a condensed version of the profile matrix. The condensed version is a vector

$$\vec{e} = (m_1, \ldots, m_m, d_1, \ldots, d_d)$$

composed of an extended set of metric values aggregated across the entire iteration, such as the total time spent in that iteration. The values $m_1, \ldots, m_m$ represent the directly measured metrics, whereas $d_1, \ldots, d_d$ represent the derived metrics, together comprising the full set of metrics listed in Table 1. The idea behind the condensed profile is that $\vec{e}$ still

carries enough information to characterize the performance behavior of an iteration accurately enough although it no longer refers to individual call paths, thus eliminating the need to compare full matrices $S^i_{c,m}$. Instead differences with respect to individual call paths are approximated by adding more specific derived metrics. At the same time, the condensed profile lowers the runtime overhead by making the distance calculation much more efficient and enables more conclusive clustering decisions by drastically reducing the dimensionality of the distance function ($|M| + |D|$ usually $\ll |C| * |M|$).

The distance between two vectors $\vec{e}_1$ and $\vec{e}_2$ is then calculated using the Manhattan distance. Whenever a new cluster is created from a fresh iteration profile, we calculate the distance between the new cluster and those that already exist. In addition, we need to calculate the distance between a cluster formed via merging and all remaining clusters, overall still leading to linear computational complexity $O(n)$ in terms of the maximum number of clusters n. Note that the calculation of derived metrics has to be done at most twice for every new iteration (i.e., once for every new cluster). In contrast, the space complexity is $O(n^2)$ because we need to store the distance between every pair of clusters.

Since metrics may have different domains and their values may cover different ranges of magnitude, we face the question of how much weight to assign to an individual metric when calculating distances. Table 1 shows how Scalasca metrics are organized in tree hierarchies defined by subset relationships, with general metrics as tree roots and more specific metrics as leaves in the tree (e.g., total time → execution time → MPI time → MPI communication time → MPI collective communication time). On the one hand, the subset relationship implies that more general metrics have higher values than more specific metrics. On the other hand, more specific metrics are typically more indicative of performance problems, such as communication overhead. We therefore decided to assign metrics equal weights and normalize the vector elements accordingly.

As reference value for the normalization, we chose the process-local running average for all preceding iterations because it is relatively stable with respect to noise-induced outliers. Where necessary, the average over all iterations could be obtained from a prior run, and can be expected not to vary substantially in the presence of minor run-to-run variations. The first distance calculation is performed only after the desired number of clusters has been reached, therefore a fair number of iterations have passed before the current average is determined for the first time. In principle, the first distance calculation could be delayed even further, depending on the availability of memory to hold iteration profiles representing unmerged clusters. Although the use of the running average may lead to premature distance calculations, which would be hard to redo without incurring substantial runtime overhead, we show in Section 5 that the inaccuracy resulting from this limitation has little influence on the fidelity of our compression. We therefore believe that in most cases a single measurement will be sufficient.

4.3 Emphasizing the Baseline

The algorithm discussed so far has a serious shortcoming. It emphasizes noise and extrema much more than it characterizes the baseline behavior. For example, in some applications the algorithm might blur periodic low-amplitude changes in the baseline by merging their constituent clusters, while preserving a number of prominent but noise-induced extrema that are distinct enough to have dedicated clusters reserved for them. The problem occurs whenever the distance between extrema is large compared to the distances within the small-scale repetitive pattern. However, baseline changes often carry valuable information on general performance trends and are therefore desirable to keep, whereas extrema often turn out to be irreproducible noise and a minor contribution to overall performance.

To better accentuate the baseline in comparison to extrema, we exploit the fact that the number of iterations that stand out is usually much smaller than the number of iterations on the baseline level. Using a heuristic, we distort the distance metric in such a way that the distance among larger clusters becomes greater, whereas the distance between smaller clusters with only a few elements becomes smaller. Two small clusters are then much more likely to be merged than two larger clusters representing many iterations. Using this method, distances between clusters remain valid until one of the clusters is merged with another one, leaving the complexity linear in terms of the maximum number of clusters.

4.4 Call-Tree Equivalence

The algorithm explained so far is based on an elastic distance criterion, allowing iterations with very different call trees to be merged. This property may allow the occurrence of *phantom call paths* in the compressed profile, which are call paths associated with an iteration, although they have never been visited during this iteration. Since phantom call paths may lead to wrong conclusions, they are a serious problem and should be avoided whenever possible. Furthermore, call-tree equivalence between two iterations is often a good indicator of similar performance characteristics. For these two reasons, we only merge iterations with equivalent call trees. Call-tree equivalence between two iterations i_1 and i_2 can be defined in two ways:

- Weak equivalence: every call path visited in i_1 has also been visited in i_2 and vice versa.
- Strong equivalence: every call path has been visited as many times in i_1 as it has been visited in i_2.

Weak equivalence excludes phantom call paths, while strong equivalence also considers quantitative similarity. To enforce call-tree equivalence between clusters before they are merged, we partition the clusters into call-tree equivalence classes. Then every new iteration profile either forms a new class or is added to an existing one, depending on whether there is already a cluster whose call tree is equivalent to the new one. If the maximum number of clusters is exceeded, the two clusters closest to each other that are located within the same partition are merged.

Since call-tree equivalence is no longer an elastic criterion, the number of equivalence classes is not configurable. Especially when strong call-tree equivalence is used, the total number of clusters may exceed the predefined threshold, resulting in a large number of partitions each with only a single cluster, increasing storage requirements and also potentially degrading the compression fidelity as distance-based clustering is then suppressed. On the other hand, if their number is small enough, enforcing call-tree equivalence can

Application	Execution time (s)	Iterations	Call paths total	Call paths per iteration min.	Call paths per iteration avg.	Call paths per iteration max.	MPI per iteration min.	MPI per iteration avg.	MPI per iteration max.	Call-tree equiv. classes weak	Call-tree equiv. classes strong
104.milc	765	-	-	-	-	-	-	-	-	-	-
107.leslie3d	847	2,000	30,802	15	15.4	17	4	4.2	5	2	**2**
113.GemsFDTD	1,051	1,000	46,001	46	46.0	47	1	1.0	2	2	**2**
115.fds4	427	2,363	86,346	36	36.5	45	2	2.1	5	13	**19**
121.pop2	242	2,000	182,233	84	91.1	103	20	23.0	28	3	**11**
122.tachyon	3,905	-	-	-	-	-	-	-	-	-	-
126.lammps	930	500	38,397	76	76.8	96	36	36.9	56	23	**24**
127.wrf2	1,282	1,375	158,075	111	115.0	489	3	3.1	37	**7**	348
128.GAPgeofem	376	235	5,641	24	24.0	25	8	8.	8	2	**7**
129.tera_tf	1,217	942	10,369	11	11.0	18	4	4.0	8	2	**2**
130.socorro	391	20	49,820	933	2491.0	2609	28	77.4	81	8	19
132.zeusmp2	998	200	22,399	111	112.0	112	56	56.	56	2	**2**
137.lu	554	180	2,881	16	16.0	17	7	7.0	8	2	**2**
PEPC	13,643	1,300	67,470	51	52.0	66	28	29.0	35	**4**	1298

Table 2: Application characteristics: execution times and iteration, call-path and equivalence class counts

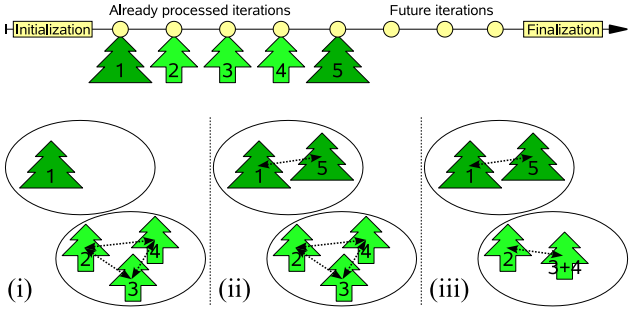


Figure 2: Incremental on-line clustering of iteration call-tree profiles into a maximum of four clusters. (i) The call-tree profiles for the first four iterations are stored directly, yet distinguished into two equivalence classes. (ii) The call-tree profile for iteration 5 is matched to the equivalence class of iteration 1. (iii) The pair of clusters with the shortest separation distance (here 3 and 4) are merged to retain only the desired number of clusters.

improve the compression fidelity with respect to call-tree structure and count-based metrics, as we will see in Section 5. Whether and to which degree call-tree equivalence should be enforced is therefore highly application dependent. A decision must be made based on the total number of call-tree equivalence classes, which would have to be determined from a prior measurement. To avoid this extra measurement, a dynamic scheme can be employed that starts with strong equivalence and switches to weak equivalence after re-grouping the existent clusters if the number of partitions becomes too large in comparison to the desired number of clusters, thus keeping the number of clusters in check. Since in our test cases the number of weak equivalence classes never exceeded 23 (as will be seen in Table 2), we expect at least weak equivalence to be a viable option for most applications.

While enforcing call-tree equivalence adds the cost of comparing potentially large call trees to determine the equivalence class of an iteration (i.e, $O(log(n))$ comparisons for n existent classes), it also reduces the number of distance calculations because distances are calculated only within each class. The performance disadvantage therefore diminishes as the number of clusters is increased. Figure 2 illustrates the basic steps of our algorithm for one process using a maximum number of four clusters.

4.5 Reconstructing Aggregate Profiles

Although our algorithm performs a lossy compression, we can accurately reconstruct an aggregate profile that summarizes across all iterations. Adding up all clusters weighted by the number of iterations they represent will yield an exact profile without any errors introduced by the lossy compression, as if obtained without phase instrumentation. While small differences between an original and a reconstructed summary profile may be caused by clustering overhead, compression errors manifest only when considering individual iterations.

5. EVALUATION

5.1 Measurements Used

For evaluation purposes, we focus on the applications of the SPEC MPI2007 benchmark suite [3, 24] in the medium-sized reference configuration with 32 MPI processes on a dedicated 32-core IBM p5-575 SMP node. Execution characteristics of the 13 applications are summarized in Table 2, along with the *PEPC* application run on an IBM Blue Gene/P rack that will be considered in Section 6. Only 10 of the 13 applications in the suite were evaluated, as in two of the applications no suitable main loop could be identified for phase instrumentation (due to code complexity for *104.milc* and the absence of such a loop for the farming-based *122.tachyon*), and one application (*130.socorro*) had too few iterations in the supplied test case. In the medium-sized reference configuration, the *121.pop2* application executes 9,000 iterations, however, we reduced this to 2,000 iterations due to memory limitations for storing measurement data because our reference data sets were obtained without compression, as explained in Section 5.2.

In addition to the wall-clock execution time and number of iterations (or timesteps) in the main computational loop of each application, full summary measurements with

instrumentation distinguishing each iteration allow analysis of the call-path complexity, which is included in Table 2. Call paths for each iteration are further distinguished to consider only those ending with MPI communication and synchronization functions, and minimum, average and maximum call-path count statistics calculated. (The number of call paths included in a summary profile depends on function in-lining by the compiler and filtering applied during measurement [3].) Furthermore, by enforcing call-tree equivalence among the sets of call paths, the number of classes resulting from weak and strong equivalence were determined. Where strong equivalence resulted in an excessive number of equivalence classes, weak equivalence was used instead (i.e., for *127.wrf2* and *PEPC*). The SPEC MPI2007 benchmark suite and the *PEPC* application are seen to have a rich variety of execution and call-tree characteristics.

Based on the numbers of iterations and call-tree equivalence classes found in these application executions, we chose to compare clusterings using powers of two from 8 to 256. Eight clusters offer a relatively small storage overhead but require aggressive compression, and can be expected to only coarsely represent complex execution characteristics. On the other hand, 256 clusters should provide improved fidelity, but at a corresponding storage cost factor for the results.

5.2 Off-line Version

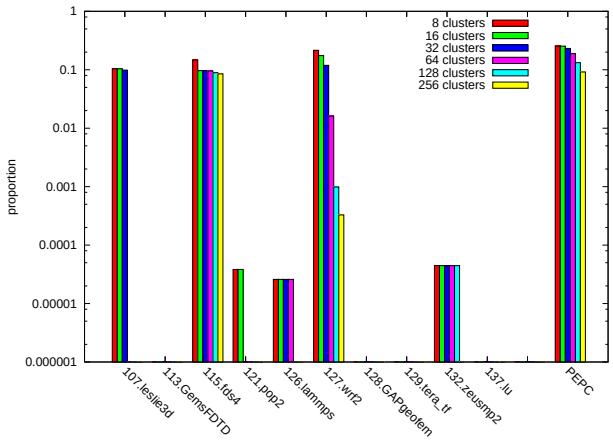
Evaluating the quality of the compressed data based on the on-line approach would not be feasible by simply taking measurements with and without compression and comparing the results. Differences due to run-to-run variation could not be distinguished from those due to the lossiness of the compression. Additionally, accurate time and memory usage measurements of the algorithm could not be taken while it ran together with the actual application measurement. Evaluation is therefore based on an off-line version of the algorithm which works on already collected full phase-instrumented measurement results. The compression algorithm is applied to this input data to create measurement results which are equivalent to those that would be collected by the on-line version of the algorithm. We ran the algorithm on the same system where the original measurements were collected to make timings comparable. The effectiveness of different configurations of the compression algorithm can also be readily compared, as they are all based on the same real measurement data.

5.3 Quality Assessment

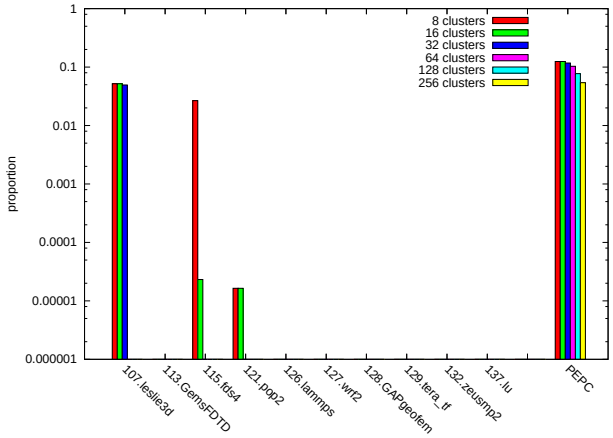
A variety of quality characteristics were investigated for *PEPC* and the SPEC MPI2007 applications with different numbers of clusters. Due to space limitations, only the most important and illustrative assessments are shown and evaluated, with discussion of results from associated analysis concisely summarized.

Erroneous call paths

All execution call paths are accurately captured when call-tree equivalence is enforced, however, without it call paths can be erroneously associated to clusters of iterations where they do not actually occur by the merging operation on distinct call trees. Figure 3 shows that there are indeed significant numbers of 'phantom' call paths added for *107.leslie3d*, *115.fds4*, *127.wrf2* and *PEPC*. While four of the applications never had 'phantom' call paths introduced, the remaining three applications each had a few erroneous call paths (sometimes only with smaller numbers of clusters), which are also potentially a serious concern. Although many of the 'phantom' call paths are not MPI call paths, and therefore less of a concern, 'phantom' MPI call paths are found for *107.leslie3d*, *115.fds4* and *121.pop2* with smaller numbers of clusters, and predominate for *PEPC* even with 256 clusters. Since the call trees often differ, and can differ significantly especially at low cluster counts, enforcing call-tree equivalence is therefore essential for accurate time-series call-path profiling. In the remainder of our evaluation, call-tree equivalence is therefore always enforced. The equivalence relation was determined based on the number of resulting classes, as indicated by the bold numbers in the rightmost two columns of Table 2. In most cases, we were able to apply strong equivalence. Only for *127.wrf2* and *PEPC* did we have to resort to weak equivalence.



(a) Call paths added.



(b) MPI call paths added.

Figure 3: Proportion of erroneously added 'phantom' call paths in reconstructed profiles when call-tree equivalence is not enforced during clustering.

Error rates for entire iterations

In this subsection, we evaluate error rates with respect to the mean and maximum values shown in the 2-dimensional iteration graphs (blue and red bars in Figure 1(a)), consid-

ering only aggregate metric values for entire iterations. We skip the discussion of the minimum values (green bars), as they are rarely influenced by outliers and therefore usually suffer the smallest compression-related distortion among the three. Figure 4 shows the error rates of mean values, averaged across iterations and metrics. Error rates for individual metrics are normalized using the average value from all iterations. Average error rates are high for *115.fds4* and *PEPC* with only a few clusters, but improve, as larger numbers of clusters are used, down to around 1% and 3% respectively with 64 clusters, while the other applications have negligible errors in their mean values. Errors in maximum values for iterations are generally only marginally worse, though *PEPC* is again an exception with an average error for the maximum metric values reaching 10% for 64 clusters and 6% for 256 clusters. Average error rates for count-based metrics are considerably better than those for time-based metrics, since call-path visits and bytes transfered are usually not subject to noise and small measurement variations such as the time-based metrics. For *PEPC*, however, the count error rates remain comparably high.

Error rates for individual call paths

Even though the call-tree equivalence verification ensures that no false call paths are reported in the data reconstructed from the compressed representations, errors are still inherent in the metric values reconstructed for every true call path. In this subsection, we therefore evaluate the quality of the compressed data at the level of individual call paths. The metric values we consider refer to a call path exclusively, that is, they do not cover its children. For each data point in the profile, the magnitude of the error can be determined from the difference of values between the full-fidelity and reconstructed call-path profiles. To emphasize the most significant call paths, the error rates under the same combination of process and metric are normalized using the maximum across all iterations and call paths under that combination. Figure 5 shows the average error rates of time-based metrics. Error rates are generally low (well below 0.1%) especially with larger numbers of clusters which suggests good characterization, and while *129.tera_tf* is seen to be particularly error prone with small numbers of clusters, it improves rapidly with more clusters. *129.tera_tf* has a continuously changing performance behavior, as evident in Figure 1. This makes it a much more challenging case than the other SPEC MPI2007 applications. As before, the count-based metrics (not shown) have even lower error rates, and a zero error rate for 8 of the 10 applications (notably including *129.tera_tf*), apart from *PEPC* where error rates are somewhat higher than for the time-based metrics.

Quantized call-path error rates

In spite of low average error rates at the call-path level, individual data points of the reconstructed profile may still exhibit significant deviations from the original data. To assess the likelihood of higher deviations, we histogrammed the normalized error rates for individual data points in buckets for orders of magnitude. Figure 6 summarizes the results for the case using 64 clusters. (Call paths which actually have zero-valued metrics are not included in this comparison to avoid dilution.)

With 64 clusters, only three of the studied applications have any error in their count-based metrics, and while less

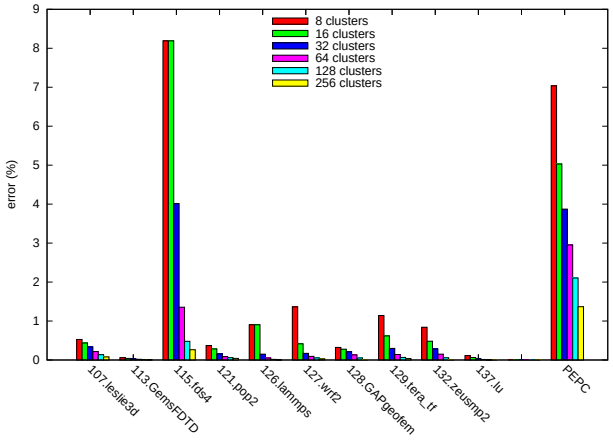


Figure 4: Average error rates of mean values of all metrics for entire iterations.

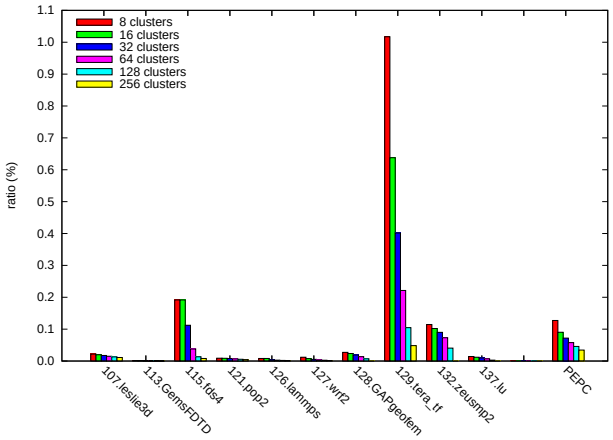
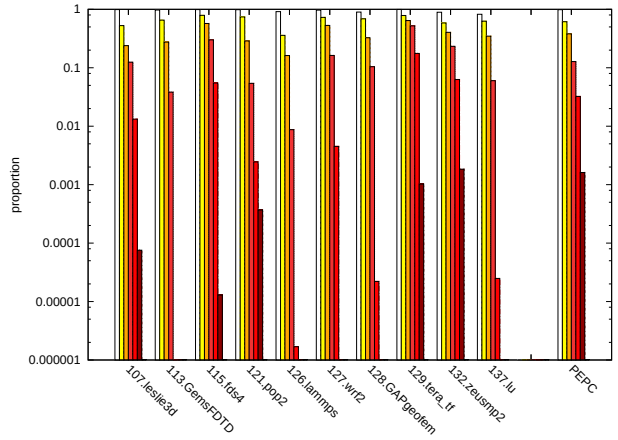
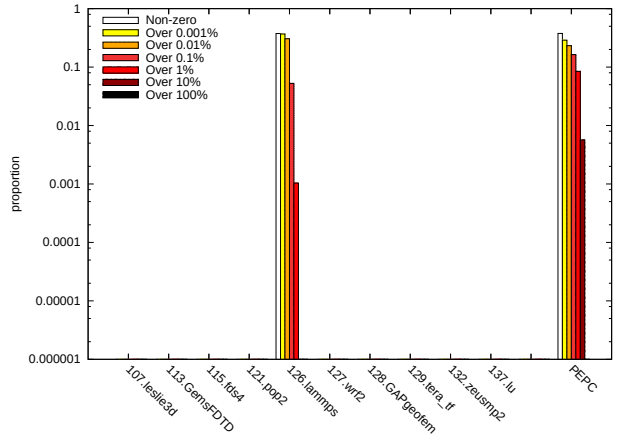


Figure 5: Average error rate of time-based metrics for each call path.

than one in a million *127.wrf2* call paths have errors of more than 0.1% (which is negligible), for *126.lammps* around one in every thousand call paths have errors exceeding 1%, and with *PEPC* around one in every two hundred call paths have more than 10% error. The *127.wrf2* errors only occur on non-MPI call paths, where they are less of a concern. However, the count-based errors in the latter two applications are serious since they are for MPI metrics and around 10% of all call paths are affected by such errors to a lesser degree. These same two applications are further distinguished from the others in sometimes having marginally lower errors in the time-based metrics, which is unusual. Entirely error-free reconstructions are rare for time-based metrics, and call-path error rates over 10% for the time-based metrics are found for *PEPC* and half of the SPEC MPI2007 applications. Such large errors affect less than one in every thousand call paths in these applications, with MPI call paths somewhat more frequently impacted. The largest call-path errors are found to have a magnitude of 64% for *PEPC* and around 33% for *121.pop2* and *132.zeusmp2*. On the other hand, all applications (except *113.GemsFDTD*) have call paths with more than 1% error in the time-based metrics, which affects one in



(a) MPI time-based metrics.



(b) MPI count-based metrics.

Figure 6: Proportion of call paths in profiles reconstructed from 64 clusters having quantized error rates for MPI time-based and count-based metrics.

five of *129.tera_tf* MPI call paths. Larger numbers of clusters can be employed to reduce the magnitude and frequency of these errors.

5.4 Cluster Processing Overheads

The cost of determining call-tree equivalence and clustering costs were measured on the same systems used when running the applications themselves, so that they can be related to the average duration of iterations to indicate processing overhead that would be introduced. Figure 7 shows that on average this overhead is around 6% for *121.pop2*, around 1% for *115.fds4* and *127.wrf2*, and less than 0.5% for *PEPC* and the other SPEC MPI2007 applications. Overheads for particular iterations and processes were found to be twice as large as the average overhead. Where application iterations are short and the call tree relatively large, processing time overhead therefore becomes considerable. While the processing time increases only slightly for larger numbers of clusters, storage requirements for the cluster distances grow as $O(n^2)$, but remain less than 4MB with 256 clusters and are therefore negligible.

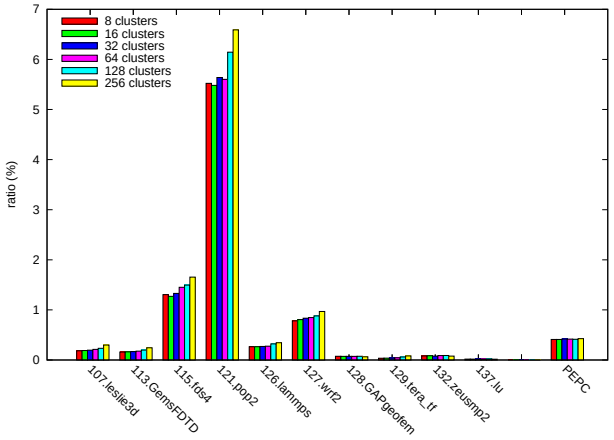


Figure 7: Average clustering time as a proportion of iteration execution time.

6. DETAILED EXAMPLE

As a large scale example, we chose the *PEPC* Coulomb-solver run with 1024 processes on an IBM Blue Gene/P, in which we were able to find the root causes of a serious performance problem using our phase-instrumentation approach [4]. It is an adaptive code, re-balancing the workload after every iteration, which makes its time-dependent behavior crucial in understanding its performance. This also makes *PEPC* much harder to characterize than the previously considered SPEC MPI2007 applications, in that its performance is completely different in each process, its baseline for many metrics increases continually, and the few bottleneck processes with very different communication behavior from all the others are constantly relocated by the computational load-balancing algorithm. (As it turned out from our analysis, the algorithm balancing the computational load is highly effective, but makes the communication load increasingly imbalanced over time.)

It is therefore no surprise that the quality analysis and comparison in the previous section show *PEPC* to be much more susceptible to 'phantom' call paths (Fig. 3), especially 'phantom' MPI call paths, and substantial error rates requiring 256 or more clusters for reasonable accuracy of both entire-iteration aggregate metric values (Fig. 4) and individual call-path exclusive metric values (Figs. 5 and 6). On the other hand, even with 256 clusters, processing overheads are reasonable (Fig. 7).

Figure 8 gives an impression of the quality of the data produced by different maximum cluster counts. It shows the iteration graphs and value maps of two of the key metrics, MPI point-to-point communication count and time reconstructed from compressed data with a selection of different cluster counts, and compares these to the full-fidelity originals. (Related graphs and maps for 8, 32 and 128 clusters are omitted for lack of space.) This shows that *PEPC* is a good example of an application where the count-based metric graphs and value maps (in the leftmost two columns) are complex, yet the reconstructions are surprisingly good and with 32 or 64 clusters all the main features of the originals are already present.

MPI point-to-point communication time metric graphs and value maps (in the rightmost two columns), however,

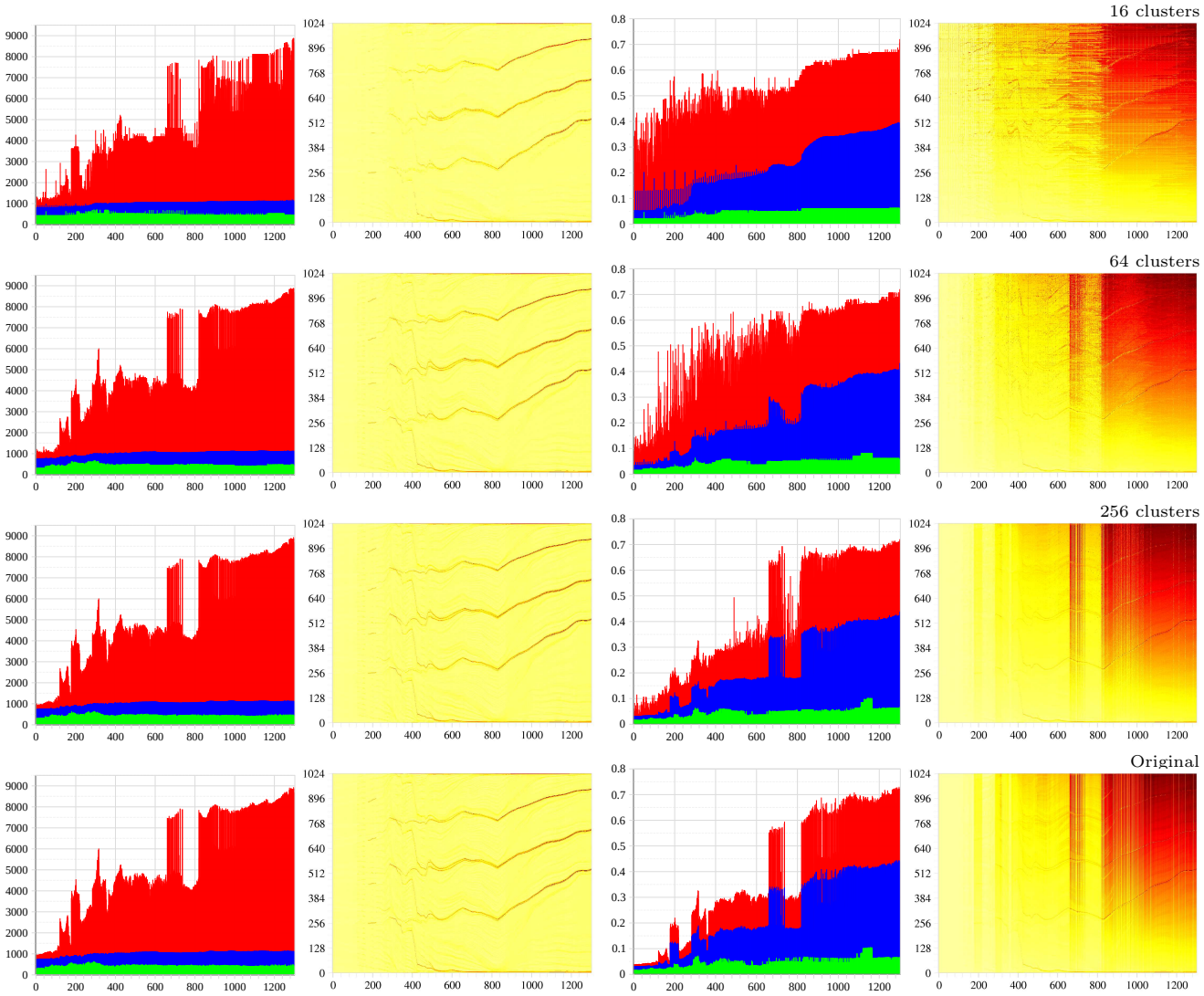


Figure 8: Comparison iteration graphs and value maps of MPI point-to-point communication count (left) and MPI point-to-point time (right) for *PEPC* with reconstructions from different cluster counts.

show that 64 clusters are clearly insufficient to reconstruct this application execution behavior. While 128 clusters was relatively close (especially for the average values), 256 clusters are needed for a really good characterization of the maximum values. The maximum value is difficult to characterize perfectly, as a single outlier on any of the processes changes the maximum for a given iteration. The value map of the same metric reconstructed from 128 clusters was also very close to the original, however, and the important features needed to find the performance problem in this application (the dark diagonal lines and the growth of the point-to-point communication time with the process rank) are already clearly visible with 64 clusters.

Figure 9 compares the MPI time profile of the full 1300-iteration *PEPC* analysis report with a reconstruction from 128 clusters. Aggregate metric values for entire timesteps are in broad agreement, with up to 5% error for timestep 1292. Even the detailed call-path profiles show good agreement with the selected MPI_Waitany call path in the penultimate timestep 1299 having only 9% error. The distribution of the selected call-path metric values over the 1024 processes of the Blue Gene topology has also been reasonably characterized, although significant differences for certain processes are evident.

At least 256 clusters therefore seem to be required to characterize all important features of *PEPC*. This means that we need less than 20% of the buffer space to store the call-path profile data at run-time, and still get sufficiently high quality data. This is especially important for *PEPC* on a Blue Gene machine with limited per-core memory: the developers naturally optimize their code using as much of the machine's memory as possible, leaving only a small amount for Scalasca measurement data.

7. CONCLUSION

A method for time-series call-path profiling was outlined and evaluated based on incremental clustering of call trees with similar metric values for space-efficient storage. While

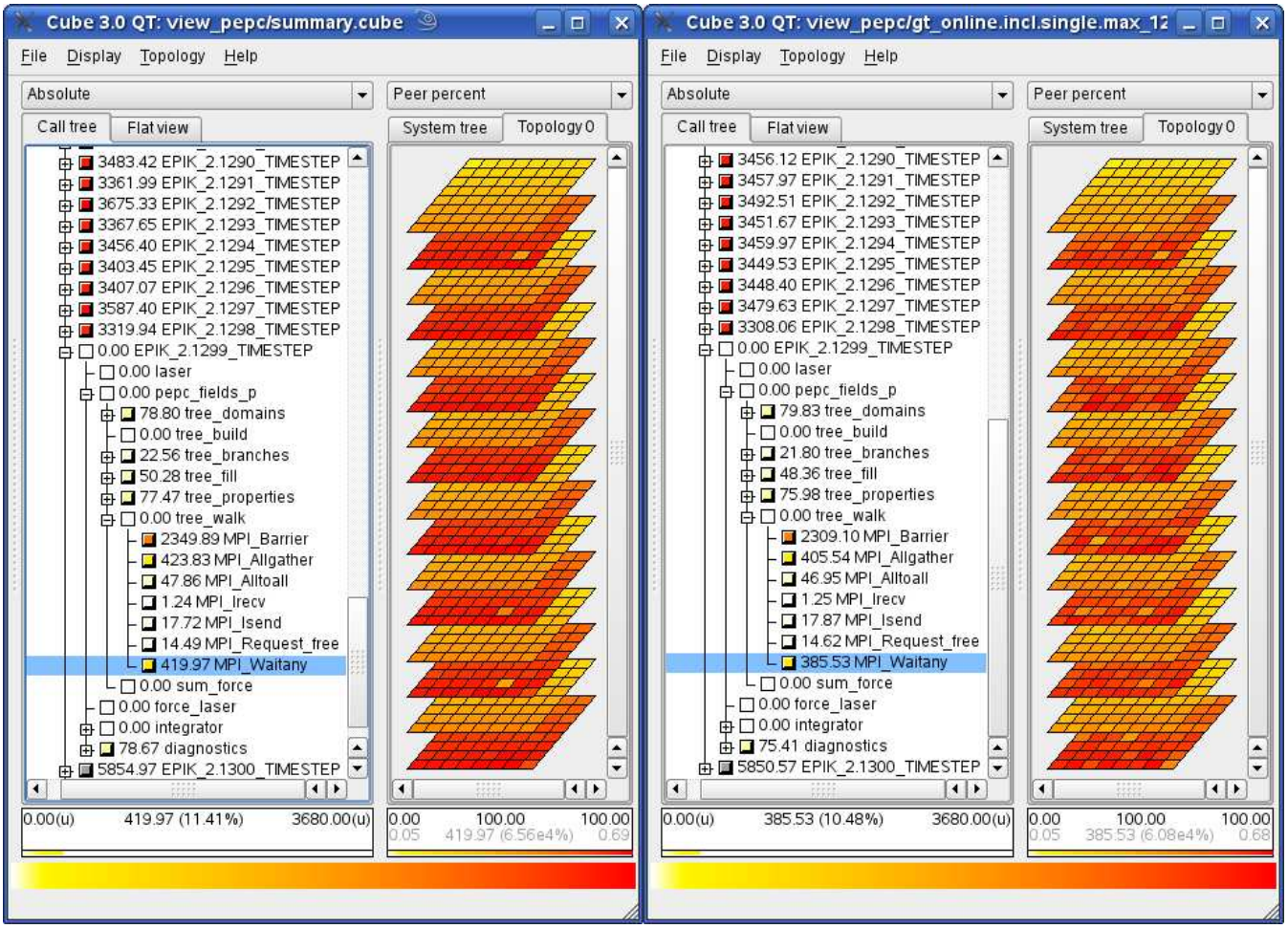


Figure 9: Scalasca analysis reports of the full *PEPC* measurement (left) and that reconstructed from 128 clusters (right), showing the MPI time metric, and with the call path selected for MPI_Waitany in the `tree_walk` routine of the penultimate timestep.

the use of a distance function based on aggregate metric values allows efficient clustering decisions, accuracy of the compressed call-path profiles can be ensured by enforcing call-tree equivalence based on visited call paths and heuristics that reduce the impact of extrema. An implementation was demonstrated with acceptable processing time and memory overheads.

Whereas the call-tree and execution characteristics of SPEC MPI2007 applications *113.GemsFDTD*, *121.pop2*, and *137.lu* lend themselves to accurate representation with as few as 8 clusters, the more complex applications *115.fds4* and *129.tera_tf* are found to need 128 clusters (with 32 or 64 clusters sufficient for the others). For the local *PEPC* application, however, at least 256 clusters are required for good time-series call-path characterization of its complicated execution behavior. 64 clusters therefore generally seems to be a reasonable default setting for a first measurement, as this value is usually high enough to ensure good-quality clustering, and still has relatively low time and memory requirements. If considered insufficient, measurements can be repeated with a larger number of clusters.

On-line application of our method will open the possibility of time-series call-path profile analysis of long-running codes consisting of thousands of iterations or timesteps. While this will offer benefits for many important applications, as demonstrated by 10 of the 13 SPEC MPI2007 benchmark codes, *121.pop2* and *PEPC* are particular examples where it will become practical to measure and analyze full-length executions for the first time.

With on-line implementation it will be necessary to study the variation of processing costs on each process in each iteration, and associated execution perturbation. Where global synchronizations by the application are identified, these may be exploited for intermediate analysis processing with reduced execution disruption. It is also desirable to automatically determine the required number of clusters, and whether enforcing call-tree equivalence can be safely omitted to lower processing overheads. OpenMP and non-MPI metrics, such as from hardware counters, will also be incorporated to enrich the analysis.

Acknowledgment

Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through Grant GSC 111 and the Helmholtz Association of German Research

Centers through Grant VH-NG-118 is gratefully acknowledged.

8. REFERENCES

[1] D. J. Kerbyson, K. J. Barker, and K. Davis, "Analysis of the weather research and forecasting (WRF) model on large-scale systems," in *Proc. of the Conference on Parallel Computing (ParCo, Aachen/Jülich, Germany)*, ser. Advances in Parallel Computing, vol. 15. IOS Press, September 2007, pp. 89–98.

[2] S. Shende, A. Malony, A. Morris, S. Parker, and J. Davison de St. Germain, "Performance evaluation of adaptive scientific applications using TAU," in *Proc. of the International Conference on Parallel Computational Fluid Dynamics (Washington DC, USA)*, May 2005.

[3] Z. Szebenyi, B. J. N. Wylie, and F. Wolf, "SCALASCA parallel performance analyses of SPEC MPI2007 applications," in *Proc. of the 1st SPEC Int'l Performance Evaluation Workshop (SIPEW, Darmstadt, Germany)*, ser. Lecture Notes in Computer Science, vol. 5119. Springer, June 2008, pp. 99–123.

[4] ——, "Scalasca parallel performance analyses of PEPC," in *Proc. of the EuroPar Workshop on Productivity and Performance (PROPER 2008, Las Palmas de Gran Canaria, Spain)*, ser. Lecture Notes in Computer Science, vol. 5415. Springer, August 2008, pp. 305–314.

[5] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A call graph execution profiler," in *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, SIGPLAN Notices*, vol. 17, no. 6, June 1982, pp. 120–126.

[6] T. Ball and J. R. Larus, "Efficient path profiling," in *Proc. of the 29th annual ACM/IEEE International Symposium on Microarchitecture (MICRO, Paris, France)*. IEEE Computer Society, 1996, pp. 46–57.

[7] L. A. DeRose and F. Wolf, "CATCH – A call-graph based automatic tool for capture of hardware performance metrics for MPI and OpenMP applications," in *Proc. of the 8th Euro-Par Conference (Euro-Par, Paderborn, Germany)*, ser. Lecture Notes in Computer Science, vol. 2400. Springer, August 2002, pp. 167–176.

[8] A. D. Malony, S. S. Shende, and A. Morris, "Phase-based parallel performance profiling," in *Proc. of the Conference on Parallel Computing (ParCo, Malaga, Spain)*, ser. NIC Series, vol. 33. John von Neumann Institute for Computing, September 2005, pp. 203–210.

[9] A. R. Bernat and B. P. Miller, "Incremental call-path profiling," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 11, pp. 1533–1547, 2007.

[10] K. Fürlinger, M. Gerndt, and J. Dongarra, "On using incremental profiling for the performance analysis of shared memory parallel applications," in *Proc. of the 13th Euro-Par Conference (Euro-Par, Rennes, France)*, ser. Lecture Notes in Computer Science, no. 4641. Springer, August 2007, pp. 62–71.

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X, San Jose, CA, USA)*. ACM, 2002, pp. 45–57.

[12] J. S. Vetter and D. Reed, "Managing performance analysis with dynamical statistical projection pursuit," in *Proc. of the Supercomputing Conference (SC1999, Portland, OR, USA)*. IEEE Computer Society, November 1999.

[13] O. Y. Nickolayev, P. C. Roth, and D. A. Reed, "Real-time statistical clustering for event trace reduction," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 144–159, 1997.

[14] D. H. Ahn and J. S. Vetter, "Scalable analysis techniques for microprocessor performance counter metrics," in *Proc. of the Supercomputing Conference (SC2002, Baltimore, MD, USA)*. IEEE Computer Society, November 2002.

[15] K. A. Huck and A. D. Malony, "PerfExplorer: A performance data mining framework for large-scale parallel computing," in *Proc. of the Supercomputing Conference (SC2005, Seattle, WA, USA)*. IEEE Computer Society, November 2005.

[16] A. Knüpfer and W. E. Nagel, "Construction and compression of complete call graphs for post-mortem program trace analysis," in *Proc. of the International Conference on Parallel Processing (ICPP, Oslo, Norway)*. IEEE Computer Society, June 2005, pp. 165–172.

[17] M. Casas, R. M. Badia, and J. Labarta, "Automatic phase detection of MPI application," in *Proc. of the Conference on Parallel Computing (ParCo, Aachen/Jülich, Germany)*, ser. Advances in Parallel Computing, vol. 15. IOS Press, September 2007, pp. 129–136.

[18] T. Gamblin, R. Fowler, and D. A. Reed, "Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes," in *Proc. of the 22nd Int'l Parallel and Distributed Processing Symposium (IPDPS, Miami, FL, USA)*. IEEE Computer Society, 2008.

[19] C. Lu and D. A. Reed, "Compact application signatures for parallel and distributed scientific codes," in *Proc. of the Supercomputing Conference (SC2002, Baltimore, MD, USA)*. IEEE Computer Society, November 2002.

[20] Scalasca: Scalable parallel performance analysis of large-scale applications, http://www.scalasca.org/.

[21] 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)*, ser. Advances in Parallel Computing, vol. 15. IOS Press, September 2007, pp. 645–652.

[22] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, 2nd ed. Morgan Kaufmann, 2006.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[24] Standard Performance Evaluation Corporation. (2007) SPEC MPI2007 benchmark suite. [Online]. Available: http://www.spec.org/mpi2007/