2011 IEEE International Parallel & Distributed Processing Symposium

Performance Analysis of Long-running Applications

Zoltán Szebenyi[*‡], Felix Wolf[*†‡], Brian J. N. Wylie[*]

[*] Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany
[†] German Research School for Simulation Sciences, 52062 Aachen, Germany
[‡] RWTH Aachen University, 52056 Aachen, Germany
{z.szebenyi, f.wolf, b.wylie}@fz-juelich.de

***Abstract*—With the growing complexity of supercomputing applications and systems, it is important to constantly develop existing performance measurement and analysis tools to provide new insights into application performance characteristics and thereby help scientists and engineers utilize computing resources more efficiently. We present the various new techniques developed, implemented and integrated into the Scalasca toolset specifically to enhance performance analysis of long-running applications. The first is a hybrid measurement system seamlessly integrating sampled and event-based measurements capable of low-overhead, highly detailed measurements and therefore particularly convenient for initial performance analyses. Then we apply iteration profiling to scientific codes, and present an algorithm for reducing the memory and space requirements of the collected data using iteration profile clustering. Finally, we evaluate the complete integration of all these techniques in a unified measurement system.**

I. Introduction

Supercomputers play a key role in countless areas of science and engineering, enabling the development of new insights and technological advances that were previously inconceivable. The strategic importance and ever-growing complexity of the efficient usage of supercomputing resources makes parallel performance analysis tools invaluable for the scientific and engineering community. The Scalasca toolset [1] is a highly scalable, open source profiling and tracing tool supporting measurements of MPI, OpenMP and hybrid MPI/OpenMP applications that has been demonstrated to effectively scale to 294,912 processes [2]. In the course of this thesis project several improvements to the Scalasca toolset were developed, implemented and evaluated to extend its applicability to an even wider range of use cases, and provide advanced features that give more insight into the complex performance phenomena encountered in long-running, large-scale applications.

Table I shows the set of representative scientific codes studied, consisting of the SPEC MPI 2007 suite of large applications complemented with the local *DROPS* and *PEPC* applications. (*PEPC* run with 1,024 processes on the *Jugene* Blue Gene/P, and the others with 256 processes on the *Juropa* Nehalem cluster.) These applications are written in a variety of languages with varying complexity, particularly in the use of MPI, and run at a range of scales on different HPC systems at Jülich Supercompuing Centre. Some perform thousands of iterations (or time-steps), others only hundreds, and in a couple of cases no clear iteration loop was identifiable (such as the *122.tachyon* ray-tracing graphics application).

II. Combining Sampling and Event-based Measurements

While the event-based instrumentation and measurement approach provided by Scalasca has been found to be effective for most applications, in some cases it suffers from prohibitively high overheads. When there are many tiny functions which are frequently executed, as common in C++ codes, the time spent recording those events can exceed the time spent in the actual user code. Selective measurement filtering and instrumentation can reduce or eliminate such cases, however, it requires scoring a preliminary measurement run and needs to be done with care. A measurement infrastructure that is more convenient to use for performance monitoring and analysis of production runs of scientific codes is desirable.

One solution to this problem is to switch to only recording statistical data about the execution, as provided by sampling-based measurements, as in [3], [4] and other tools. An interval timer is configured to periodically interrupt execution to acquire snapshots of the current state of execution. While in an event-based measurement the current context is tracked continuously, with the position in the call tree updated at each subsequent enter and exit event, a disadvantage of sampling-based techniques is that we lose track of the context between samples. To still be able to build the dynamic call tree of the application, we have to locate our position in the call tree

Application	Execution time [s]	Iterations	Max. call paths per iteration All	Max. call paths per iteration MPI	Call-tree equiv. classes
121.pop2lref	461	440	543	228	10
122.tachyon	484	-	-	-	-
125.RAxML	1003	21,268	316	63	18
126.lammps	574	300	458	78	8
128.GAPgeofem	575	2,501	55	17	11
129.tera_tf	281	190	89	19	3
132.zeusmp2	248	200	183	91	2
137.lu	269	180	67	29	3
142.dmilc	203	-	-	-	-
143.dleslie	254	15,054	38	12	4
145.lGemsFDTD	326	1,500	317	29	2
147.l2wrf2	683	720	1730	110	6
DROPS	497	50	18675	475	29
PEPC	13,643	1,300	66	35	4

TABLE I
Application characteristics: execution times and iteration, call-path and equivalence class counts

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.388

IEEE computer society

at every sample event. Call stack unwinding is employed to determine the current call path. Since unwinding the stack is an expensive operation, a variety of optimizations were necessary including maintaining a thunk stack containing architecture-specific jump instructions.

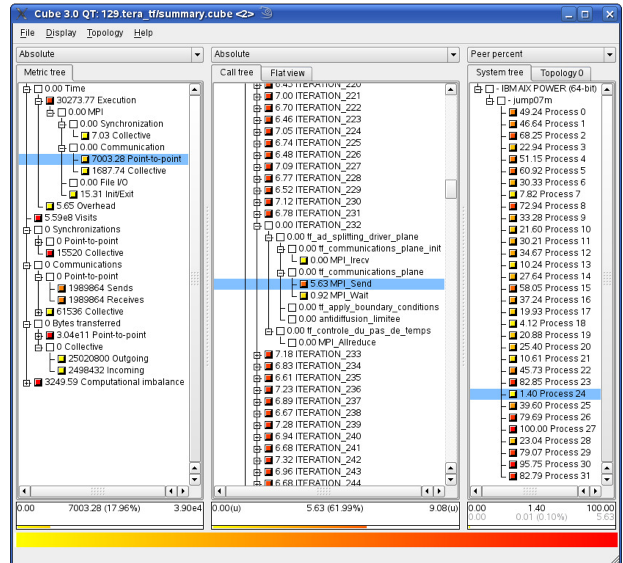
Another limitation of sampling-based measurements is that they provide only statistically approximate data not just from user functions, but also from MPI events like communication function calls. As the communication characteristics of the application are especially relevant when analyzing parallel efficiency and scalability, recording all the communication events is preferable. Moreover, access to communication function arguments is straightforward with standard PMPI wrappers but generally not possible from call-stack samples, which renders the measurement system unable to collect many important communication metrics like “Number of bytes transferred.”

To circumvent these problems, we developed a hybrid combination of sampling and event-based measurements, where sampling is used for user code and direct instrumentation is applied to communication events. In this way, we get the best of both techniques, combining full information about communication events with low overhead from sampling of the user code. A similar hybrid approach is presented in [5], collecting the two kinds of measurement data separately in profiles and traces and merging them after measurement. In contrast, our system provides a sophisticated, seamless integration of the two measurement types, paying close attention to details such as sample interrupts inside communication events and sample intervals that contain one or more MPI calls. We evaluated the technique using the SPEC MPI 2007 applications, and demonstrated its usefulness by pinpointing actual scalability bottlenecks in the *DROPS* C++ fluid dynamics code [6].

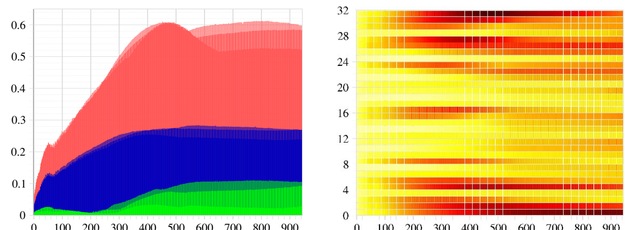
III. Iteration profiling

Call-path profiling, 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. For example, in the analysis of MPI programs, call-path information for each process is often essential to determine where in the program a communication bottleneck occurs. However, these kinds of measurements only give the user a summary overview of the entire execution, without regard to changes in performance behavior over time. As scientific applications tend to be run for extended periods of time, simply neglecting these changes is no longer sufficient, and understanding the patterns and trends in the performance data along the time dimension becomes crucial.

Most scientific applications have an easily identifiable main loop, e.g., iterating through discrete timesteps of a simulation. To get a better understanding of the temporal changes of the performance characteristics of these applications, a simple source code instrumentation interface was introduced to mark the beginning and end of the main loop, enabling Scalasca to collect a separate profile for each iteration. There is ongoing work looking at the possibilities of identifying recurring phases automatically [7], but for our purposes simple manual



(a) 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).



(b) Graph of MPI point-to-point communication time per iteration with maximum (red), median (blue), and minimum (green) process values.

(c) Value map of MPI point-to-point communication time per iteration for each process with higher values shown darker.

Fig. 1. Different views for analyzing the time-dependent behavior of a 32-process *129.tera_tf* experiment from the SPEC MPI 2007 suite with Scalasca.

instrumentation similar to that presented in [8] was sufficient. This extension made it possible to generate more fine-grained measurement reports, and results in a wealth of invaluable insights not possible before. As demonstrated in our study of the SPEC MPI 2007 benchmark suite [9], there is a wide variety of performance behavior patterns to be explored in the time dimension, ranging from periodic or irregular peaks in communication or computation metrics, through gradually changing execution times in subsequent iterations, to sudden transitions of the base-line behavior. Moreover, performance behavior is often subject to significant process-dependent variations, i.e., the performance behavior can be a function of both time and space as demonstrated in Fig. 1.

An interesting example of this is shown in our study [10] of plasma simulation with *PEPC* — an adaptive code that re-balances workload after each time-step. In an execution

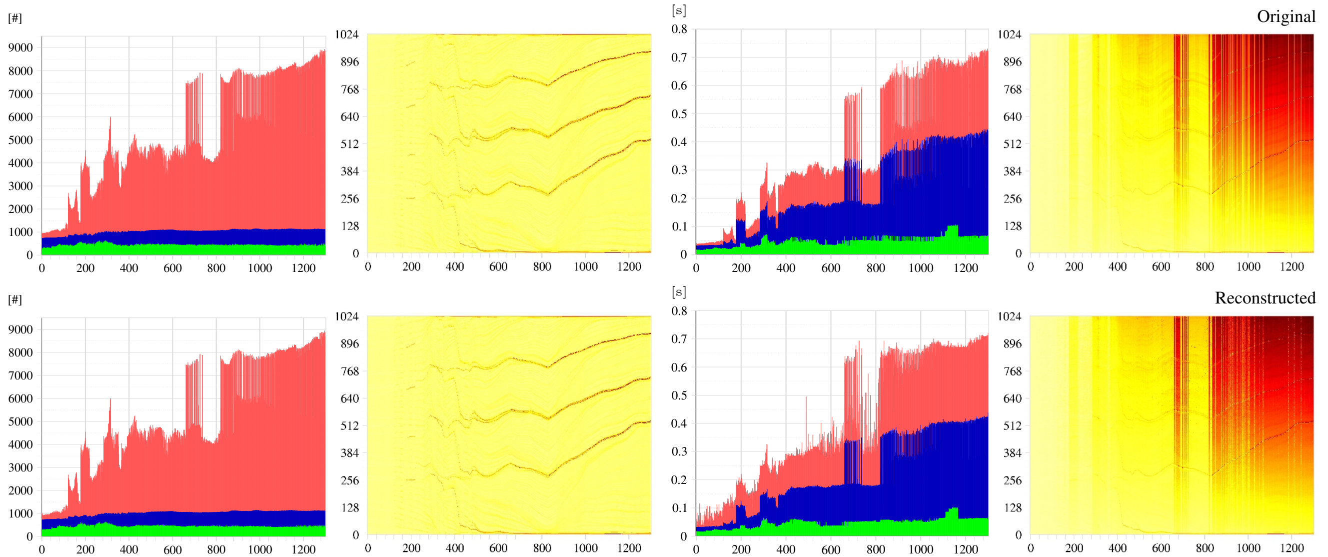


Fig. 2. Comparison of the original, uncompressed iteration graphs and value maps of MPI Point-to-point communication count (left) and time (right) with those reconstructed from 256 profile clusters for 1,300 timesteps of *PEPC* execution with 1,024 processes on BG/P.

using 1,024 processes on IBM Blue Gene/P, the performance bottleneck was caused by a gradually growing communication imbalance on only a handful of processes, but the bottleneck moved on to neighboring processes after each time-step due to workload re-balancing. As seen in the upper part of Fig. 2, over the course of 1,300 timesteps, as *PEPC* continually re-distributes particles to balance the computational work between the 1,024 processes, a few processes rapidly acquire very large numbers of particles. The number of MPI point-to-point communications and the associated communication time grow proportionally to the number of particles a process is responsible for. Since a one-dimensional list of processes is employed, the hot-spots with the largest numbers of particles gradually migrate to adjacent processes as the simulation evolves. A simple profile without the time dimension has limited value, as it only shows that the bottleneck was present on every process but not that it was not present on all at the same time and moving to new processes every iteration.

IV. Iteration profile clustering

A significant problem with iteration instrumentation is that the introduction of the time dimension makes the amount of data collected proportional to the number of iterations, and memory usage and file sizes can become impractical. This is a serious limitation, as memory space tends to be at a premium on highly parallel machines like IBM Blue Gene, and the increased file sizes can lead to expensive, prolonged report writing times. Also, post-processing and report examination typically happens on standard desktop machines with limited memory and storage capacity, and become impractical when working with very large amounts of data.

However, it is clear that there is a huge amount of redundancy inherent in these measurement results, as subsequent iterations of the same code loop often have very similar call trees and very similar overall performance characteristics. Thus, it is a waste of resources to collect every iteration separately. Based on these observations, a compression algorithm was developed to exploit the redundancy. The growing importance of the problem is shown by the fact that other groups have also looked at the compression of performance data, using compressed complete call graphs [11] or a clustering-based method compressing data along the process dimension [12]. The requirements for our algorithm were quite restrictive: it had to be very low overhead, both in time and storage, as it is used during measurement where keeping our footprint at a minimum is a must. It had to be an on-line algorithm, taking decisions on-the-fly as data arrives, and adapting to changes in the performance behavior dynamically. This is because we can't collect all data before starting to compress it, since we want to keep the all-time memory usage of the measurement system at a minimum. Some of the data could be compressed in a lossy way, as minor differences from the original performance metrics can be tolerated, but other aspects such as the call-tree structure of each iteration had to be retained exactly as they occurred.

The algorithm is based on the idea of incremental clustering, collecting the metric profiles of similar iterations to the same cluster — and of dissimilar ones into separate clusters — as they are generated while measurement of the target application progresses. As the temporal patterns in the applications' performance characteristics can show strong process-dependent variations, clustering is done separately on each process. This has the added benefit that no synchronization or communication between the processes is necessary at run time. The algorithm is lossy in the sense that the compressed data no longer contains all the information necessary to restore the original data, however, our evaluations show that the result of

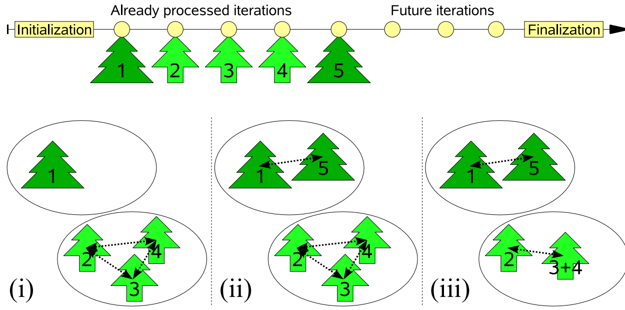


Fig. 3. Incremental on-line clustering of iteration call-tree profiles into a maximum of four clusters. (i) The call-tree profiles for the first four iterations are stored directly, yet distinguished into two equivalence classes. (ii) The call-tree profile for iteration 5 is matched to the equivalence 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.

the reconstruction comes very close to the original data.

Simple, direct control over the tradeoff between resource usage and compression quality is possible by specifying the desired cluster count. As long as the cluster count is not reached, each iteration is stored as a separate cluster. When the desired cluster count would be surpassed, the most similar clusters are merged to keep the cluster count constantly at the desired level, as illustrated in Fig. 3. Clusters associated with more than one iteration store the average of the performance data from those iterations over all call paths of their call trees. So-called "phantom call paths," that were not executed in that iteration but only in other iterations of the same cluster, would result in confusing analysis. These are avoided by only merging iterations with identical call trees. The distance function used to determine the similarity of two clusters is the Manhattan distance of the vectors of overall metric values, containing a single normalized entry for each metric. Much more information would be available if we compared values on every call path of the call trees separately, but to keep overheads low and avoid the curse of dimensionality, we restrict the comparison to the overall values. Also, a heuristic function is applied to the distance computation that makes the merging of clusters with high element counts increasingly less likely. This is because exact data for one-off events, which are likely to be noise, is far less interesting than the detailed representation of those phenomena that occur most often. Our evaluations show that applying this heuristic greatly increases the fidelity of our final compressed data.

Clustering with 256 profile clusters and the reconstruction of individual time-step iterations for the challenging example use case provided by the *PEPC* plasma simulation code is shown in the lower part of Fig. 2. Note the slight compression quality difference between the communication count (left) and time (right) metrics: in general, exact representation of time-based metrics is not possible, due to their susceptibility to noise and irregular variations, while even lower cluster counts tend to be enough for an almost perfect representation of count-based metrics, even in complex cases. Most applications have much simpler behavior, where even better compression ratios can be achieved while still maintaining high compression quality.

V. Complete integration

Finally, the techniques were integrated producing a measurement system capable of the collection, analysis and compression of time-dependent performance data collected using a hybrid approach based on both call-stack sampling and instrumented events. The hybrid approach's ability to collect exact communication metrics was important, as it helps directing the algorithm's decisions to create the best clustering. Also, given the stochastic nature of sampling, new call-tree structure matching algorithms had to be developed, comparing only the call paths leading to communication calls (which are still deterministic), since exact comparison of the statistical data we have about the user call paths would be meaningless. The integrated solution is currently being evaluated applying all the above features together to a representative set of applications, and will be available in a forthcoming release of Scalasca.

References

[1] Scalasca. Scalable performance analysis of large-scale parallel applications. [Online]. Available: http://www.scalasca.org/

[2] B. J. N. Wylie, M. Geimer, B. Mohr, D. Böhme, Z. Szebenyi, and F. Wolf, "Large-scale performance analysis of Sweep3D with the Scalasca toolset," *Parallel Processing Letters*, vol. 20, no. 4, pp. 397–414, Dec. 2010.

[3] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A call graph execution profiler," *SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[4] N. Froyd, J. Mellor-Crummey, and R. Fowler, "Low-overhead call path profiling of unmodified, optimized code," in *Proc. 19th Int'l Conf. on Supercomputing (ICS, Cambridge, MA, USA)*. ACM, 2005, pp. 81–90.

[5] A. Morris, A. D. Malony, S. Shende, and K. Huck, "Design and implementation of a hybrid parallel performance measurement system," in *Proc. 39th Int'l Conf. on Parallel Processing (ICPP, San Diego, USA)*. IEEE Computer Society, September 2010, pp. 492–501.

[6] Z. Szebenyi, F. Wolf, B. J. N. Wylie, T. Gamblin, M. Schulz, and B. R. de Supinski, "Reconciling sampling and direct instrumentation for the performance analysis of parallel programs," in *Proc. 25th Int'l Parallel and Distributed Processing Symposium (IPDPS, Anchorage, AK, USA)*. IEEE Computer Society, 2011.

[7] M. Casas, R. M. Badia, and J. Labarta, "Automatic phase detection of MPI applications," 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.

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

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

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

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

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