# Chapter 2
# Using Sampling to Understand Parallel Program Performance

**Nathan R. Tallent and John Mellor-Crummey**

**Abstract** Developing scalable parallel applications for extreme-scale systems is challenging. The challenge of developing scalable parallel applications is only partially addressed by existing languages, compilers, and autotuners. As a result, manual performance tuning is often necessary to obtain high application performance. Rice University's HPCTOOLKIT is a suite of performance tools that supports innovative techniques for pinpointing and quantifying performance bottlenecks in fully optimized parallel programs with a measurement overhead of only a few percent. Many of these techniques were designed to leverage sampling for performance measurement, attribution, analysis, and presentation. This paper surveys some of HPCTOOLKIT's most interesting techniques and argues that sampling-based performance analysis is surprisingly versatile and effective.

**Keywords** Performance tools • Sampling • Call path profiling • Call path tracing • HPCTOOLKIT

## 2.1 Introduction

Because of the complexities of modern microprocessor-based supercomputers—e.g., scale, hierarchical parallelism, heterogeneity, deep memory hierarchies—typical scientific applications achieve only a small fraction of a supercomputer's peak performance. As a result, manual performance analysis and tuning is often

N.R. Tallent (✉)
Pacific Northwest National Laboratory, Richland, WA 99352, USA
e-mail: tallent@pnnl.gov

J. Mellor-Crummey
Department of Computer Science, Rice University, Houston, TX 77005, USA
e-mail: johnmc@rice.edu

H. Brunst et al. (eds.), *Tools for High Performance Computing 2011*,
DOI 10.1007/978-3-642-31476-6_2, 

necessary to identify and resolve performance bottlenecks to avoid wasting precious computational resources.

Rice University's HPCTOOLKIT [1, 19] is a suite of performance tools that has pioneered techniques for pinpointing and quantifying performance bottlenecks in fully optimized parallel programs with a measurement overhead of only a few percent [2, 5, 9, 13, 15, 22–24, 26–29]. Many of these techniques were designed to leverage sampling for performance measurement, attribution, analysis, and presentation. This paper surveys some of HPCTOOLKIT's most interesting techniques and argues that sampling-based performance analysis is surprisingly versatile and effective.

Prima facie, sampling is both an obvious and questionable technique for performance analysis. Consider the challenge of collecting both accurate and detailed performance measurements. There are two basic techniques for collecting performance measurements: asynchronous sampling and synchronous instrumentation. Although there is an inverse relationship between accuracy and precision, the elasticity of that relationship varies with measurement technique. With instrumentation, collecting context-sensitive measurements at the level of a procedure can easily lead to hundreds or even thousands of percent overhead [3, 9, 23]; measuring at finer resolutions such as the loop, statement, or machine-instruction level is often infeasible. Efforts to address these problems include statically [4, 10, 21] or dynamically [12, 16, 21] omitting instrumentation, which leads to blind spots; adjusting measurements based on a model of how instrumentation affects execution [14]; and other specialized techniques [3, 11, 30]. We argue that, when possible, it is better to simply avoid instrumentation overhead and its associated distortion. To provide a more elastic trade-off between accuracy and precision than instrumentation can provide, HPCTOOLKIT collects statistically representative measurements by periodically interrupting an execution using an asynchronous event trigger. Just as pollsters query a thousand voters to discern trends in a population of 300 million, asynchronously sampling the activity in a program execution can provide representative information about the program's behavior.

But if sampling appears an obvious choice, there are a number of considerations that make it questionable. First, the fact that asynchronous sampling interrupts an execution at arbitrary points poses certain difficulties. One difficulty is how to attribute measurements to source-level loops. With source code instrumentation such attribution is trivial (though costly). With sampling it is difficult because loops in object code are implemented using unstructured control flow and have been transformed by optimizing compilers. Another difficulty is collecting calling contexts using stack unwinding. Experience shows that unwinds commonly fail when using libunwind [17] or GLIBC's `backtrace()` routine [7] from within highly optimized code. Second, the largest supercomputers present challenges of their own. To avoid idleness in bulk synchronous parallel applications, supercomputer operating systems frequently attempt to minimize OS 'jitter' [18]. Unfortunately from the perspective of sampling, a standard way to minimize jitter is to forbid frequent software interrupts. Supercomputer kernels may also leave unimplemented

the OS support needed to drive asynchronous sampling triggers. Third, it is often not clear how to use sampling to accomplish a primary goal of performance analysis: pinpointing root causes of bottlenecks. For instance, because sampling may miss the first (or any other) link in a chain of events, it may have limited diagnostic powers. More concretely, consider the problem of pinpointing the causes of lock contention. A straightforward solution uses instrumentation to time lock acquisitions and releases and attributes those measurements to their calling context. However, adding such instrumentation will lengthen critical sections and often exacerbate lock contention. Is it possible to use a sampling-based technique that can determine, for only a few percent of overhead, when a thread is working with a lock, working without a lock, or waiting for a lock? More to the point, is it possible to not just identify threads that wait for locks, but the threads and critical sections that *cause* waiting?

This paper is organized as follows. Section 2.2 summarizes HPCTOOLKIT's call path profiling capabilities. Section 2.3 explains how to pinpoint scaling bottlenecks using differential analysis. Section 2.4 presents techniques for pinpointing the causes (not effects) of bottlenecks. Section 2.5 describes how to use sampling for collection and presentation of call path traces. Section 2.6 summarizes HPCTOOLKIT's data-centric attribution capabilities. Finally, Sect. 2.7 summarizes our conclusions. Because of space constraints and a retrospective perspective, we delegate a thorough discussion of related work to individual research papers.

## 2.2 Call Path Profiling

Experience shows that comprehensive performance analysis of modern modular software requires information about the full *calling context* in which costs are incurred. For instance, the costs incurred for calls to communication primitives (e.g., `MPI_Wait`) or code that results from instantiating C++ templates for data structures can vary widely depending upon their calling context. Consequently, HPCTOOLKIT uses asynchronous sampling to collect *call path profiles* that attribute costs to the full calling contexts in which they are incurred.

Collecting a call path profile requires capturing the calling context for each sample event. To capture the calling context for a sample event, HPCTOOLKIT's measurement tool `hpcrun` must be able to unwind the call stack at *any* point in a program's execution. Obtaining the return address for a procedure frame that does not use a frame pointer is challenging since the frame may dynamically grow (space is reserved for the caller's registers and local variables; the frame is extended with calls to `alloca`; arguments to called procedures are pushed) and shrink (space for the aforementioned purposes is deallocated) as the procedure executes. To cope with this situation, we developed a fast, on-the-fly binary analyzer that examines a routine's machine instructions and computes how to unwind a stack frame for the procedure [23]. For each address in the routine, there must be a recipe for how to unwind. Different recipes may be needed for different intervals of addresses within

the routine. Each interval ends with an instruction that changes the state of the routine's stack frame. Each recipe describes (1) where to find the current frame's return address, (2) how to recover the value of the stack pointer for the caller's frame, and (3) how to recover the value that the base pointer register had in the caller's frame. Once we compute unwind recipes for a routine, we memoize them for later reuse.

To apply our binary analysis to compute unwind recipes, we must know where each routine starts and ends. When working with applications, one often encounters partially stripped libraries or executables that are missing information about function boundaries. To address this problem, we developed a binary analyzer that infers routine boundaries by noting instructions that are reached by call instructions or instructions following unconditional control transfers (jumps and returns) that are not reachable by conditional control flow.

HPCTOOLKIT's use of binary analysis for call stack unwinding has proven to be very effective, even for fully optimized code. A detailed study of our unwinder on versions of the SPEC CPU2006 benchmarks optimized with several different compilers showed that the unwinder was able to recover the calling context for all but a vanishingly small number of cases [23].

One particularly useful and novel feature of HPCTOOLKIT's call path profiles is that they are enriched with static source code structure. Such call path profiles are constructed by overlaying `hpcrun`'s dynamic contexts with static source code structure computed post mortem by the `hpcstruct` tool. In an off-line process called *recovering program structure*, this tool constructs an object to source code mapping. As a result, HPCTOOLKIT's call path profiles expose loops and inlined frames and attribute metrics to them—all for an average overhead of 1–5 % [23].

## 2.3 Pinpointing Scaling Bottlenecks

We now present an elegant and powerful way to apply HPCTOOLKIT's sampling-based call path profiles to identify scalability bottlenecks in Single-Program Multiple-Data (SPMD) applications, whether executed on a multicore node or a leadership-class supercomputer. The basic idea is to compute a metric that quantifies scaling loss by scaling and differencing call path profiles from a pair of executions [5, 26].

Consider two parallel executions of an application, one executed on $p$ processors and the second executed on $q > p$ processors. In a weak scaling scenario, processors in each execution compute on the same size data. If the application exhibits perfect weak scaling, then the execution times should be identical on both $p$ and $q$ processors. In fact, if every part of the application scales uniformly, then this equality should hold in each corresponding part of the application.

Using HPCTOOLKIT, we collect call path profiles on each of $p$ and $q$ processors to measure the cost associated with each calling context in each execution. HPCTOOLKIT's `hpcrun` profiler uses a data structure called a *calling context tree*

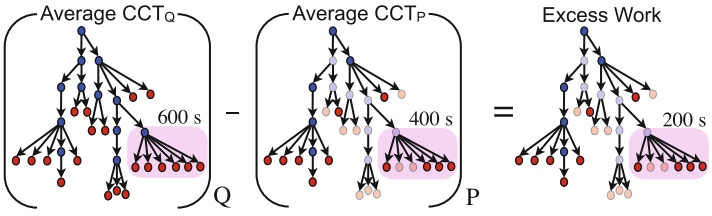


**Fig. 2.1** Differencing call path profiles to pinpoint scaling bottlenecks

(CCT) to record a call path profile. Each node in a CCT is identified by a code address. In a CCT, the path from any node to the root represents a calling context. Each node has a weight $w \geq 0$ indicating the exclusive cost attributed to the path from that node to the root. Given a pair of CCTs, one collected on $p$ processors and another collected on $q$ processors, with perfect weak scaling, the cost attributed to all pairs of corresponding CCT nodes should be identical. Any additional cost for a CCT node on $q$ processors when compared to its counterpart in a CCT for an execution on $p$ processors represents *excess work*. This process is shown pictorially in Fig. 2.1. The fraction of excess work, i.e., the amount of excess work in a calling context in a $q$-process execution divided by the total amount of work in a $q$-process execution represents the scalability loss attributed to that calling context. By scaling the costs attributed in a CCT before differencing them to compute excess work, one can also use this strategy to pinpoint and quantify strong scalability losses [5]. As long as the CCTs are expected to be similar, this analysis strategy is independent of the programming model and bottleneck cause.

To enable this simple but powerful scalability analysis on the largest class of supercomputers—and to pave the way for other sampling-based tools such as Open|SpeedShop [20] and CrayPat [6]—the HPCTOOLKIT team engaged kernel developers at IBM and Cray to address shortcomings in their microkernels that prevented any sort of sampling-based measurement [26]. We have since applied our scalability analysis to pinpoint synchronization, communication, and I/O bottlenecks in applications on several large-scale distributed-memory machines. Because our analysis is based on differencing comparable executions, we have also used it to pinpoint and quantify scaling bottlenecks on multicore nodes at the loop level [25].

## 2.4 Blame Shifting

Standard sampling-based measurement identifies symptoms of performance losses rather than causes. Although some symptoms may have an obvious cause, simply knowing symptoms is frequently not sufficient to justify a concrete set of actions for resolving a bottleneck. For example, consider the case of a parallel run time system with idle worker threads. A standard call path profile will show that the

idle threads spend a large percentage of their execution time in a scheduling loop waiting for work. The cause of these threads' idleness is that at least some parts of the application are insufficiently parallel in that they fail to generate enough parallel tasks to keep other worker threads busy. But which parts? Instead of reporting that a worker thread frequently idles, it would be better if a tool pinpointed source code that should generate more parallel tasks to keep idle threads busy. We call the process of redirecting attribution from symptoms to causes *blame shifting*.

In general, blame shifting uses either in situ or post mortem analysis to shift the attribution of metrics from performance symptoms to their causes. The need for blame shifting is patently obvious because the primary goal of performance tuning is to find and resolve bottlenecks. However, it is a challenge to make both precise and accurate causal inferences while inducing minimal measurement perturbation. This section discusses three types of blame shifting employed by HPCTOOLKIT.

### *2.4.1 Parallel Idleness and Overhead in Work Stealing*

Work stealing is a popular scheduling technique for dynamic load balancing. One of the most influential versions of work stealing was implemented to support the Cilk language [8] for shared-memory parallel computing. Cilk—along with its descendants Cilk++ and Cilk Plus—combines lazy parallel task creation and a work-stealing scheduler. The Cilk run time maps logically independent asynchronous calls (tasks) onto worker threads (compute cores) in an efficient way. For a Cilk program to execute efficiently, an application must generate enough "well sized" parallel tasks. That is, there should be enough parallel tasks to keep worker threads busy, but those tasks should be large enough so that the overhead of packaging a task (i.e., creating continuations) is insignificant.

The Cilk model challenges existing performance tools. To diagnose causes—not symptoms—of inefficient execution, a tool must solve at least three problems. First, a tool must be able to not only detect when worker threads are idling but also identify the portion of the application causing idleness. Second, a tool must be able to detect when worker threads, though busy, are executing useless work. Third, a tool must be able to attribute its conclusions to source code in its calling context. Although this last challenge may seem trivial, associating costs with the context in which they are incurred is not as simple as it sounds. With a work stealing scheduler, call paths become fragmented because procedure frames migrate between worker threads to balance the work load. Consider a case where thread $y$ steals a task from thread $x$. In this case, the procedure frames for a source-level call path become separated in both space and time: space, because thread $x$ contains $y$'s parent context; time, because thread $x$ continues executing rather than blocking and waiting for thread $y$ to complete the asynchronous call. As a result, a standard call path profile of a Cilk program yields a result far from what an application developer would expect.

Our solutions to these problems build upon HPCTOOLKIT's call path profiling. To attribute metrics to source-level call paths, we developed logical call path

profiling [22, 24], a generalization of call path profiling that maps system-level to source-level call paths. To form the source-level call paths, we stitch together path fragments from a thread's stack and heap data structures.

To identify the causes of insufficient parallelism in Cilk applications, it is necessary to establish, with minimal measurement overhead, a causal link between an idling thread and a working thread that does not generate parallel work. We establish likely suspects using the following scheme [22, 24]. First we make a slight adjustment to the Cilk run-time to always maintain $W$ and $I$, the number of working and idle threads, respectively. This can be done by maintaining a node-wide counter representing $W$; since the number of worker threads $T$ is constant we have $I = T - W$. Second, we slightly modify our sampling strategy. If a sample event occurs in a thread that is not working, we ignore it. When a sample event occurs in a thread that is actively working, the thread attributes one sample to the work metric for its sample context. It then obtains $W$ and $I$ and attributes a fractional sample $I/W$ to the idleness metric for the sample context. Even though the thread itself is not idle, it is critical to understand what work the thread is performing when other threads are idle. Our strategy charges the thread its proportional responsibility for not keeping the idle processors busy at that moment at that point in the program. As an example, consider taking a sample of a Cilk execution where five threads are working and three threads are idle. Each working thread records one sample of work in its work metric, and $3/5$ sample of idleness in its idleness metric. The total amount of work and idleness charged for the sample is 5 and 3, respectively.

To identify parallel overhead in Cilk applications, we must distinguish between useful work and overhead. Our solution is based on the observation that it is possible to use static analysis to classify object code instructions as either useful work or overhead. Thus, HPCTOOLKIT can attribute a precise measure of parallel overhead to source-level calling contexts for no additional measurement overhead other than that induced by logical call path profiling [22, 24].

### 2.4.2 Lock Contention

Locking is the most common means of coordinating accesses while reading and writing mutable shared data structures in threaded programs. Thus, it is important that a performance tool pinpoint causes of lock contention.

There are at least two critical differences between performance monitoring for locks and for Cilk. First, identifying the cause of lock contention requires a more precise causal connection than with idleness in Cilk. With Cilk it is sufficient to observe that at any given instant when threads are idle, all working threads can be thought of as *equally* culpable for not generating enough parallel work to keep idle threads busy. In contrast, with locking, any number of threads may be idling because exactly one working thread holds a lock. Thus, it makes no sense to blame non-lock-holders for idleness. Second, with many lock-based programs, it is not tenable to atomically increment and decrement shared counters on every state change within

a thread. In a work stealing run time, steals are relatively infrequent. In contrast, locks may be acquired and released very frequently—and extending key critical sections with an atomic increment could introduce new bottlenecks. Because of these differences, the techniques we described for work stealing are insufficient to offer insight into lock contention. With locks, one must establish causal connections without significantly increasing critical sections and introducing a new bottleneck.

To solve both problems mentioned above, we use locks themselves as a communication channel for precise attribution [28]. For now, we limit our attention to spin (non-blocking) locks. Our technique is based on three rules. First, when a working thread receives a sample, we increment a thread-local work metric. Second, when we sample an idle thread spin-waiting for a lock, we (atomically) increment an idleness metric associated with the lock. The expected number of samples received for all threads waiting on that particular lock is proportional to the time spent waiting for that lock. Furthermore, because the expected number of samples is relatively small when compared with the total number of machine instructions executed during a long-latency spin-wait, the atomic increments are relatively inexpensive. Third, when a working thread releases a lock, we associate all idleness with the lock release point in its full calling context (i.e., as the lock holder releases the lock, it atomically swaps the idleness counter with 0). In this way, we precisely blame the lock release point and accurately attribute to it the idleness that it caused. An important observation is that it is usually straightforward to identify lock acquire points from release points.

We applied our techniques to the MADNESS quantum chemistry application. For the particular input we studied, MADNESS represented a significant challenge: it allocated a total of 65M distinct locks (which were used to implement futures), had a maximum of 340K live locks at any particular instance in the execution; and issued an average of 30K lock acquisitions per second per thread. For a monitoring overhead of a few percent, we found that the primary source of lock contention was in adding work (futures) to a shared work queue [28].

### *2.4.3 Load Imbalance*

Load imbalance is an important problem in programs based on bulk synchronous parallelism. Moreover, many load imbalance problems only appear in medium- to large-scale executions. Even though HPCTOOLKIT can collect detailed call path profiles in a scalable fashion, with profiles it is often difficult to pinpoint load imbalance. In contrast, with a trace (which collects performance information with respect to time) it is relatively easy to identify the effects and possible causes of load imbalance [31]. However, there are significant challenges to collecting fine-grained large-scale traces in a scalable fashion.

Exploiting HPCTOOLKIT's ability to collect large-scale call path profiles, we developed a post mortem analysis that identifies regions of a call path profile that cause load imbalance. Our analysis is based on several observations. First, it is

possible to divide an execution into two components: work and exposed (non-sleep) idleness. Second, idleness that occurs around synchronization points is a symptom of load imbalance. Third, the causes of load imbalance are typically reflected in work variability over processes; and for a given procedure or source line in a call path profile, one can compute statistics over all processes in an execution that measure variability. Fourth, with layered software, important imbalance and synchronization points are often context sensitive. For example, an application may have many calls to `MPI_Allreduce`. While the work between some pairs of `MPI_Allreduce` operations may be balanced, between other pairs it may not, causing lightly-loaded threads to idle. Given these observations, we developed a post mortem blame shifting analysis of call path profiles that identifies exposed idleness and associates that idleness with possible causes [27].

## 2.5 Call Path Tracing

Although tracing is a powerful performance-analysis technique, tools that employ it can quickly become bottlenecks themselves. To avoid large overheads, instrumentation-based tracing tools typically monitor only a subset of the procedures in an application. However, even coarse-grained trace data of large-scale executions quickly becomes unwieldy and a challenge to present. Unfortunately, to obtain actionable performance feedback for modular parallel software systems, it is often necessary to have fine-grained context-sensitive data—the very thing scalable tools avoid. We have developed sampling-based techniques to collect and present arbitrary slices of fine-grained large-scale call path traces [29].

Figure 2.2 shows a screen shot of HPCTOOLKIT's presentation tool `hpctraceviewer`. As with other tools that present space-time visualizations, time advances from left to right on a horizontal axis and MPI-rank space advances from top to bottom on a vertical axis. Unlike other tools, `hpctraceviewer`'s renderings are hierarchical. Since each sample in each process's time line represents a call path, we can view the process timelines at different call path depths to show more or less detailed views of the execution. The Figure shows one slice of the process/time/call-path space at depth 3; the inset shows the selected region at depths 3, 6 and 7.

To collect traces, HPCTOOLKIT exploits a useful property of our CCT data structure: a full call path can be represented by a single leaf node. Thus, we can generate a full call path trace by maintaining a CCT and stream of compact trace records where each trace record is 12 bytes (4-byte CCT node id and an 8-byte timestamp). For reasonable sampling rates, the data rate of this trace stream is extremely low. Moreover, because sampling rates are controllable, the granularity of the trace and its corresponding data volume can scale to very large or long executions.

To present traces, `hpctraceviewer` also uses sampling. Given a display window of height $h$ and width $w$ (in pixels), the tool simply samples the trace data records to determine how each pixel should be colored. Using sampling,

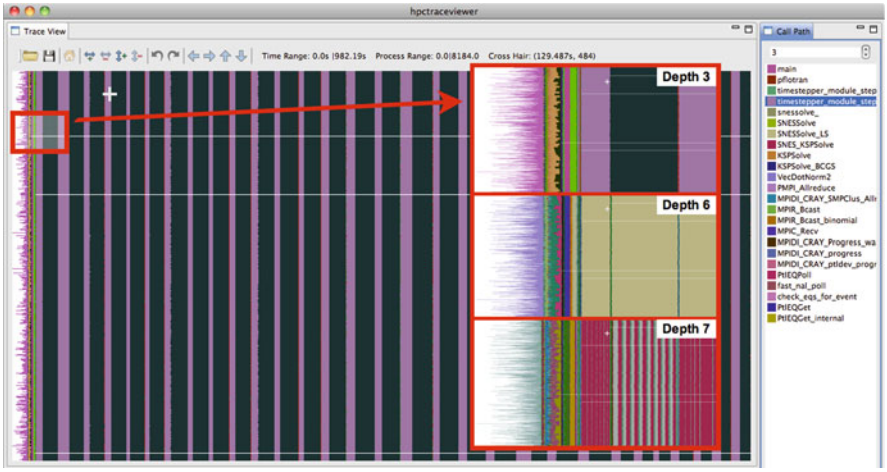


**Fig. 2.2** An 8184-core execution of PFLOTRAN on a Cray XT5. The inset exposes call path hierarchy by showing the selected region (*top left*) at different call path depths

`hpctraceviewer` can render call-path depth slices at arbitrary resolutions in time $O(hw \log t)$, where $t$ is the number of trace records in the largest trace file. In practice, the $\log t$ factor is often negligible because we use linear extrapolation to predict the locations of the $w$ trace records. The practical upshot of this technique is that `hpctraceviewer` can rapidly present trace files of arbitrary length for executions with an arbitrary number of threads on a laptop.

## 2.6 Data-Centric Performance Analysis

Sampling-based profilers such as HPCTOOLKIT attribute samples to object and source code for very low overhead. By exploiting recent hardware support, HPCTOOLKIT uses sampling to attribute measurements not only to source code but also to data objects [13]. With this *data-centric* attribution, HPCTOOLKIT can attribute metrics such as memory access latency to program data objects and their corresponding source code use points—in its full calling context. The results are especially useful when long-lived data objects are accessed at many places.

## 2.7 Conclusions

We have found that sampling-based techniques are surprisingly versatile and effective. Using sampling-based measurement, HPCTOOLKIT can collect exquisitely detailed call path profiles and traces for an average run-time overhead of less

than 5 %. Using sampling-based presentation, HPCTOOLKIT effortlessly presents arbitrary slices of large-scale fine-grained call path traces. To make sampling-based call path profiling of fully optimized applications possible, HPCTOOLKIT employs a set of innovative techniques to collect call paths at arbitrary points in an execution (using stack unwinding) and attribute to loops and inlined functions. In many cases HPCTOOLKIT's code-centric profiles are sufficient for understanding performance problems, but for more difficult problems, it is necessary to understand how aggregate costs arise over time or with respect to data. To pinpoint and quantify many scaling bottlenecks, it is sufficient to simply compare two or more call path profiles—accurate call path profiles that show statements in their full calling context and expose loops. To offer more insight into the *causes* of scaling bottlenecks, we developed several problem-focused analyses that shift blame from a bottleneck's effects to its causes or potential causes. Because they are based on sampling, these techniques incur very low measurement overhead and apply to problems as diverse as lock contention, load imbalance, insufficient parallelism in work stealing, and parallel overhead. The ability to collect and present accurate, detailed and problem-focused measurements for large-scale production executions has enabled HPCTOOLKIT's use on today's grand challenge applications: multi-lingual programs leveraging third-party libraries for which source code and symbol information are often unavailable.

In many ways, sampling is a perfect fit for performance analysis. One issue with sampling-based methods is that while they tend to show representative behavior, they might miss interesting extreme behavior. While this property may limit the use of sampling in correctness tools, we have not found it to be problematic for performance tools. The reason is that, assuming an appropriately sized and uncorrelated sample, sampling-based methods expose anomalies *that generally affect an execution*. In other words, although using sampling-based methods can miss any specific anomaly, those same methods will expose the anomaly's effects if they are important for performance. If the anomaly does not interfere with the application's execution, then it is not particularly important with respect to performance. If on the other hand, process synchronization transfers the anomaly's effects to other process ranks, we expect sampling-based methods to expose that fact.

**Acknowledgements** HPCTOOLKIT would not be what it is without the efforts of Mark Krentel, Laksono Adhianto, and Mike Fagan. Xu Liu developed our data-centric analysis.

## References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**(6), 685–701 (2010)
2. Adhianto, L., Mellor-Crummey, J., Tallent, N.R.: Effectively presenting call path profiles of application performance. In: International Conference on Parallel Processing Workshops, pp. 179–188. IEEE Computer Society, Los Alamitos (2010)

3. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 168–179. ACM, New York (2001)
4. Chung, I.H., Walkup, R.E., Wen, H.F., Yu, H.: MPI performance analysis tools on Blue Gene/L. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 123. ACM, New York (2006)
5. Coarfa, C., Mellor-Crummey, J., Froyd, N., Dotsenko, Y.: Scalability analysis of SPMD codes using expectations. In: Proceedings of the 21st International Conference on Supercomputing, pp. 13–22. ACM, New York (2007)
6. De Rose, L., Homer, B., Johnson, D., Kaufmann, S., Poxon, H.: Cray performance analysis tools. In: Tools for High Performance Computing, pp. 191–199. Springer, Berlin (2008)
7. Free Software Foundation: Glibc. http://www.gnu.org/s/libc/ (2012)
8. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 212–223. ACM, New York (1998)
9. Froyd, N., Mellor-Crummey, J., Fowler, R.: Low-overhead call path profiling of unmodified, optimized code. In: Proceedings of the 19th International Conference on Supercomputing, pp. 81–90. ACM, New York (2005)
10. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurr. Comput. Pract. Exp. **22**(6), 702–719 (2010)
11. Hollingsworth, J.K., Miller, B.P., Cargille, J.: Dynamic program instrumentation for scalable performance tools. In: Proceedings of the 1994 Scalable High Performance Computing Conference, pp. 841–850. IEEE Computer Society, Los Alamitos, CA, USA (1994)
12. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool-set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Berlin (2008)
13. Liu, X., Mellor-Crummey, J.: Pinpointing data locality problems using data-centric analysis. In: Proceedings of the 2011 IEEE/ACM International Symposium on Code Generation and Optimization, Chamonix, France, pp. 171–180. IEEE Computer Society, Los Alamitos (2011)
14. Malony, A.D., Shende, S., Morris, A., Wolf, F.: Compensation of measurement overhead in parallel performance profiling. Int. J. High Perform. Comput. Appl. **21**(2), 174–194 (2007)
15. Mellor-Crummey, J., Fowler, R., Marin, G., Tallent, N.: HPCView: a tool for top-down analysis of node performance. J. Supercomput. **23**(1), 81–104 (2002)
16. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn parallel performance measurement tool. Computer **28**(11), 37–46 (1995)
17. Mosberger-Tang, D.: libunwind. http://www.nongnu.org/libunwind (2012)
18. Petrini, F., Kerbyson, D.J., Pakin, S.: The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, p. 55. IEEE Computer Society, Washington, DC (2003)
19. Rice University: HPCToolkit performance tools. http://hpctoolkit.org (2012)
20. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open|SpeedShop: an open source infrastructure for parallel performance analysis. Sci. Program. **16**(2–3), 105–121 (2008)
21. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
22. Tallent, N.R., Mellor-Crummey, J.: Effective performance measurement and analysis of multithreaded applications. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 229–240. ACM, New York (2009)
23. Tallent, N.R., Mellor-Crummey, J., Fagan, M.W.: Binary analysis for measurement and attribution of program performance. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 441–452. ACM, New York (2009)

24. Tallent, N.R., Mellor-Crummey, J.M.: Identifying performance bottlenecks in work-stealing computations. Computer **42**(12), 44–50 (2009)
25. Tallent, N., Mellor-Crummey, J., Adhianto, L., Fagan, M., Krentel, M.: HPCToolkit: performance tools for scientific computing. J. Phys. Conf. Ser. **125**, 012088 (5pp) (2008)
26. Tallent, N.R., Mellor-Crummey, J.M., Adhianto, L., Fagan, M.W., Krentel, M.: Diagnosing performance bottlenecks in emerging petascale applications. In: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing, pp. 1–11. ACM, New York (2009)
27. Tallent, N.R., Adhianto, L., Mellor-Crummey, J.M.: Scalable identification of load imbalance in parallel executions using call path profiles. In: Proceedings of the 2010 ACM/IEEE Conference on Supercomputing, pp. 1–11. IEEE Computer Society, Washington, DC (2010)
28. Tallent, N.R., Mellor-Crummey, J.M., Porterfield, A.: Analyzing lock contention in multithreaded applications. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 269–280. ACM, New York (2010)
29. Tallent, N.R., Mellor-Crummey, J.M., Franco, M., Landrum, R., Adhianto, L.: Scalable fine-grained call path tracing. In: Proceedings of the 25th International Conference on Supercomputing, pp. 63–74. ACM, New York (2011)
30. Traub, O., Schechter, S., Smith, M.D.: Ephemeral instrumentation for lightweight program profiling. Tech. rep., Harvard University (1999)
31. Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Frings, W., Fürlinger, K., Geimer, M., Hermanns, M.A., Mohr, B., Moore, S., Pfeifer, M., Szebenyi, Z.: Usage of the Scalasca toolset for scalable performance analysis of large-scale parallel applications. In: Tools for High Performance Computing, pp. 157–167. Springer, Berlin (2008)