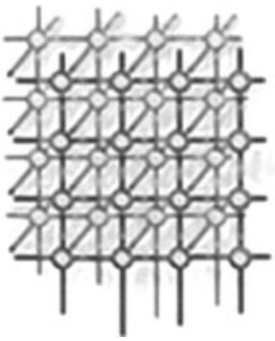
Interesting summary. Main takeaway is that any tools that instrumented code (e.g. gprof; VTune) introduce performance overheads. Since we can't trust the performance of the system (measured speed, exec time etc) we need to properly avoid instrumentation.

Otherwise, gives architectural breakdown - that's pretty much it.

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:685–701
Published online 30 December 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1553



HPCTOOLKIT: tools for performance analysis of optimized parallel programs‡

L. Adhianto[1], S. Banerjee[1], M. Fagan[1], M. Krentel[1], G. Marin[2], J. Mellor-Crummey[1,*,†] and N. R. Tallent[1]

[1]*Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251-1892, U.S.A.*
[2]*Oak Ridge National Laboratory, One Bethel Valley Road, P.O. Box 2008 MS6173, Oak Ridge, TN 37831-6173, U.S.A.*

SUMMARY

HPCTOOLKIT is an integrated suite of tools that supports measurement, analysis, attribution, and presentation of application performance for both sequential and parallel programs. HPCTOOLKIT can pinpoint and quantify scalability bottlenecks in fully optimized parallel programs with a measurement overhead of only a few percent. Recently, new capabilities were added to HPCTOOLKIT for collecting call path profiles for fully optimized codes without any compiler support, pinpointing and quantifying bottlenecks in multithreaded programs, exploring performance information and source code using a new user interface, and displaying hierarchical space–time diagrams based on traces of asynchronous call path samples. This paper provides an overview of HPCTOOLKIT and illustrates its utility for performance analysis of parallel applications. Copyright © 2009 John Wiley & Sons, Ltd.

Received 23 August 2008; Accepted 13 September 2009

KEY WORDS: performance tools; call path profiling; tracing; binary analysis; execution monitoring

1. INTRODUCTION

High-performance computers have become enormously complex. Today, the largest systems consist of tens of thousands of nodes. Nodes themselves are equipped with one or more multicore

*Correspondence to: J. Mellor-Crummey, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251-1892, U.S.A.
†E-mail: johnmc@cs.rice.edu
‡WWW: http://hpctoolkit.org

Contract/grant sponsor: Department of Energy's Office of Science; contract/grant numbers: DE-FC02-07ER25800, DE-FC02-06ER25762

microprocessors. Often, these processor cores support additional levels of parallelism, such as short vector operations and pipelined execution of multiple instructions. Microprocessor-based nodes rely on deep multi-level memory hierarchies for managing the latency and improving the data bandwidth to processor cores. Subsystems for interprocessor communication and parallel I/O add to the overall complexity of these platforms. Recently, accelerators such as graphics chips and other co-processors have started to become more common on nodes. As the complexity of HPC systems has grown, the complexity of applications has grown as well. Multi-scale and multi-physics applications are increasingly common, as are coupled applications. As always, achieving top performance on leading-edge systems is critical. The inability to harness such machines efficiently limits their ability to tackle large problems of interest. As a result, there is an urgent need for effective and scalable tools that can pinpoint a variety of performance and scalability bottlenecks in complex applications.

Nearly a decade ago, Rice University began developing a suite of performance tools now known as HPCTOOLKIT. This effort initially began with the objective of building tools that would help to guide our own research on compiler technology. As our tools matured, it became clear that they would also be useful for application developers attempting to harness the power of parallel systems. Since HPCTOOLKIT was developed in a large part for our own use, our design goals were that it be simple to use and yet provide fine-grain detail about application performance bottlenecks. We have achieved both of these goals.

This paper provides an overview of HPCTOOLKIT and its capabilities. HPCTOOLKIT consists of tools for collecting performance measurements of fully optimized executables without adding instrumentation, analyzing application binaries to understand the structure of optimized code, correlating measurements with program structure, and presenting the resulting performance data in a top-down fashion to facilitate rapid analysis. The rest of the paper is organized as six sections. Section 2 outlines the methodology that shaped HPCTOOLKIT's development and provides an overview of some of HPCTOOLKIT's key components. Sections 3 and 4 describe HPCTOOLKIT's components for measurement and analysis, respectively. Section 5 describes HPCTOOLKIT's tools for presentation of profile and trace data. In this section, we use a parallel particle-in-cell simulation of tubulent plasma in a tokamak to illustrate HPCTOOLKIT's capabilities for analyzing the performance of complex scientific applications. Section 6 briefly relates HPCTOOLKIT's approach to that of other performance tools. Section 7 offers some conclusions and sketches our plans for enhancing HPCTOOLKIT for emerging petascale systems.

2. METHODOLOGY

We have developed a performance analysis methodology, based on a set of complementary principles that, while not novel in themselves, form a coherent synthesis that is greater than the constituent parts. Our approach is *accurate*, because it assiduously avoids systematic measurement error (such as that introduced by instrumentation), and *effective*, because it associates useful performance metrics (such as parallel idleness or memory bandwidth) with important source code abstractions (such as loops) as well as dynamic calling context. The following principles form the basis of our methodology. Although we have identified several of these principles in earlier work [1], it is helpful to revisit them as they continually stimulate our ideas for revision and enhancement.

DOI: 10.1002/cpe

Be language independent: Modern parallel scientific programs often have a numerical core written in some modern dialect of Fortran and leverage frameworks and communication libraries written in C or C++. For this reason, the ability to analyze multi-lingual programs is essential. To provide language independence, HPCTOOLKIT works directly with application binaries rather than source code.

Avoid code instrumentation: Instrumentation—whether source-level, compiler-inserted or binary—can distort the application performance through a variety of mechanisms [2]. The most common problem is overhead, which distorts measurements. The classic tool `gprof` [3], which uses compiler-inserted instrumentation, induced an average overhead of over 100% on the SPEC 2000 integer benchmarks [4]. Intel's VTune [5], which uses static binary instrumentation, claims an average overhead of a factor of eight for call graph profiling. Intel's Performance Tuning Utility [6] includes a call graph profiler based on Pin's dynamic binary instrumentation [7]; we found that it yielded an average overhead of over 400% on the SPEC 2006 integer benchmarks [8].

Another problem is the tradeoff between accuracy and precision. While all measurement approaches must address this tradeoff, the problem is particularly acute for instrumentation. Source-level instrumentation can distort the application performance by interfering with inlining and template optimization. To avoid these effects, tools such as TAU intentionally refrain from instrumenting certain procedures [9]. Ironically, the more this approach reduces overhead, the more it reduces precision. For example, a common selective instrumentation technique is to ignore small frequently executed procedures—but these may be just the thread synchronization library routines that are critical. To avoid instrumentation's pitfalls, HPCTOOLKIT uses statistical sampling to measure performance.

Avoid blind spots: Production applications frequently link against fully optimized and even partially stripped binaries, e.g. math and communication libraries, for which the source code is not available. To avoid systematic error, one must measure the costs for the routines in these libraries; for this reason, source code instrumentation is insufficient. However, fully optimized binaries create challenges for call path profiling and hierarchical aggregation of performance measurements (see Sections 3 and 4.1). To deftly handle optimized and stripped binaries, HPCTOOLKIT performs several types of binary analyses.

Context is essential for understanding layered and object-oriented software: In modern, modular programs, it is important to attribute the costs incurred by each procedure to the different contexts in which the procedure is called. 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. Because often there are layered implementations within applications and libraries, it is insufficient either to insert instrumentation at any one level or to distinguish costs based only upon the immediate caller. For this reason, HPCTOOLKIT supports call path profiling to attribute costs to the full calling contexts in which they are incurred.

Any one performance measure produces a myopic view: Measuring time or only one species of event seldom diagnoses a correctable performance problem. One set of metrics may be necessary to identify a problem and another set may be necessary to diagnose its causes. For example, counts of cache misses indicate problems only if both the *miss rate* is high and the latency of the misses is not hidden. HPCTOOLKIT supports collection, correlation, and presentation of multiple metrics.

Derived performance metrics are essential for effective analysis: Typical metrics such as elapsed time are useful for identifying program hot spots. However, tuning a program usually requires a

DOI: 10.1002/cpe

measure of not where the resources are consumed, but where they are consumed *inefficiently*. For this purpose, derived measures such as the difference between the peak and the actual performance are far more useful than raw data such as operation counts. HPCTOOLKIT's `hpcviewer` user interface supports computation of user-defined derived metrics and enables users to rank and sort program scopes using such metrics.

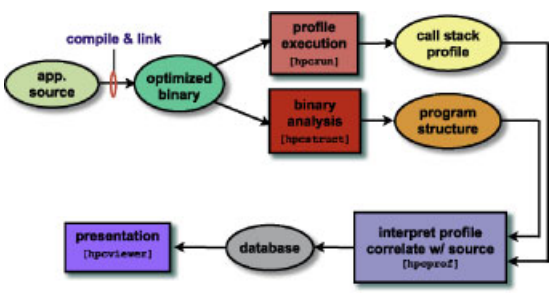
Performance analysis should be top-down: It is unreasonable to require users to wade through mountains of data to hunt for evidence of important problems. To make the analysis of large programs tractable, performance tools should present measurement data in a hierarchical fashion, prioritize what appear to be important problems, and support a top-down analysis methodology that helps users to quickly locate bottlenecks without the need to wade through irrelevant details. HPCTOOLKIT's user interface supports the hierarchical presentation of the performance data according to both static and dynamic contexts, along with ranking and sorting based on metrics.

Hierarchical aggregation is vital: The amount of instruction-level parallelism in processor cores can make it difficult or expensive for hardware counters to precisely attribute particular events to specific instructions. However, even if fine-grain attribution of events is flawed, the total event counts within loops or procedures will typically be accurate. In most cases, it is the balance of operation counts within loops that matters—for instance, the ratio between floating point arithmetic and memory operations. HPCTOOLKIT's hierarchical attribution and presentation of measurement data deftly addresses this issue; loop level information available with HPCTOOLKIT is particularly useful.

Measurement and analysis must be scalable: Today, large parallel systems may have tens of thousands of nodes, each equipped with one or more multicore processors. For performance tools to be useful on these systems, measurement and analysis techniques must scale to tens and even hundreds of thousands of threads. HPCTOOLKIT's sampling-based measurements are compact and the data for large-scale systems is not unmanageably large. Furthermore, as we describe later, HPCTOOLKIT supports a novel approach for quantifying and pinpointing the scalability bottlenecks conveniently on systems independent of scale.

2.1. From principles to practice

From these principles, we have devised a general methodology embodied by the workflow depicted in the accompanying figure. The workflow is organized around four principal capabilities:



HPCTOOLKIT workflow

DOI: 10.1002/cpe

4 Components of HPCToolkit.

1. *measurement* of performance metrics while an application executes;
2. *analysis* of application binaries to recover the program structure;
3. *correlation* of dynamic performance metrics with source code structure; and
4. *presentation* of performance metrics and associated source code.

To use HPCTOOLKIT to measure and analyze an application's performance, one first compiles and links the application for a production run, using *full* optimization. Second, one launches an application with HPCTOOLKIT's measurement tool, `hpcrun`, which uses statistical sampling to collect a performance profile. Third, one invokes `hpcstruct`, HPCTOOLKIT's tool for analyzing the application binary to recover information about files, functions, loops, and inlined code[§]. Fourth, one uses `hpcprof` to combine information about an application's structure with dynamic performance measurements to produce a performance database. Finally, one explores a performance database with HPCTOOLKIT's `hpcviewer` graphical user interface.

At this level of detail, much of the HPCTOOLKIT workflow approximates other peformance analysis systems, with the most unusual step being binary analysis. In the following sections, we outline how the methodological principles described above suggest several novel approaches to both accurate measurement (Section 3) and effective analysis (Section 4).

3. ACCURATE PERFORMANCE MEASUREMENT

This section highlights the ways in which we apply the methodological principles from Section 2 to measurement. Without accurate performance measurements for fully optimized applications, analysis is unproductive. Consequently, one of our chief concerns has been designing an accurate measurement approach that simultaneously exposes low-level execution details while avoiding systematic measurement error, either through large overheads or through systematic dilation of execution. For this reason, HPCTOOLKIT avoids instrumentation and favors *statistical sampling*.

Statistical sampling: Statistical sampling uses a recurring event trigger to send signals to the program being profiled. When the event trigger occurs, a signal is sent to the program. A signal handler then records the context where the sample occurred. The recurring nature of the event trigger means that the program counter is sampled many times, resulting in a histogram of program contexts. As long as the number of samples collected during execution is sufficiently large, their distribution is expected to approximate the true distribution of the costs that the event triggers are intended to measure.

Event triggers: Different kinds of event triggers measure different aspects of the program performance. Event triggers can be either asynchronous or synchronous. Asynchronous triggers are not initiated by direct program action. HPCTOOLKIT initiates asynchronous samples using either an interval timer or hardware performance counter events. Hardware performance counters enable HPCTOOLKIT to statistically profile events, such as cache misses and issue-stall cycles. Synchronous triggers, on the other hand, are generated via direct program action. Examples of interesting

[§] For the most detailed attribution of application performance data using HPCTOOLKIT, one should ensure that the compiler includes line map information in the object code it generates. While HPCTOOLKIT does not need this information to function, it can be helpful to users trying to interpret the results. Since compilers can usually provide line map information for fully optimized code, this requirement need not require a special build process.

DOI: 10.1002/cpe

events for synchronous profiling are memory allocation, I/O, and inter-process communication. For such events, one might measure bytes allocated, written, or communicated, respectively.

Maintaining control over parallel applications: To manage profiling of an executable, HPCTOOLKIT intercepts certain process control routines including those used to coordinate thread/process creation and destruction, signal handling, dynamic loading, and MPI initialization. To support the measurement of unmodified, dynamically linked, optimized application binaries, HPCTOOLKIT uses the library preloading feature of modern dynamic loaders to preload a profiling library as an application is launched. With library preloading, process control routines defined by HPCTOOLKIT are called instead of their default implementations. For statically linked executables, we have developed a script that arranges to intercept process control routines at link time.

Call path profiling and tracing: Experience has shown that comprehensive performance analysis of modern modular software requires information about the full *calling context* in which costs are incurred. The calling context for a sample event is the set of procedure frames active on the call stack at the time the event trigger fires. We refer to the process of monitoring an execution to record the calling contexts in which event triggers fire as *call path profiling*. To provide insight into an application's dynamic behavior, HPCTOOLKIT also offers the option to collect *call path traces*.

When synchronous or asynchronous events occur, `hpcrun` records the full calling context for each event. A calling context collected by `hpcrun` is a list of instruction pointers, one for each procedure frame active at the time the event occurred. The first instruction pointer in the list is the program address at which the event occurred. The remainder of the list contains the return address for each active procedure frame. Rather than storing the call path independently for each sample event, we represent all of the call paths for events as a calling context tree (CCT) [10]. In a CCT, the path from the root of the tree to a node corresponds to a distinct call path observed during execution; a count at each node in the tree indicates the number of times that the path to that node was sampled. Since the calling context for a sample may be completely represented by a node in the CCT, to form a trace we simply augment a CCT profile with a sequence of tuples, each consisting of a 32-bit CCT node id and a 64-bit time stamp.

Coping with fully optimized binaries: Collecting a call path profile or trace requires capturing the calling context for each sample event. To capture the calling context for a sample event, `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 [8]. 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 in 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 all intervals in a routine, we memorize 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

DOI: 10.1002/cpe

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. At present, HPCTOOLKIT provides binary analysis for stack unwinding on the x86_64, Power, and MIPS architectures. A detailed study of the x86_64 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 [8].

Handling dynamic loading: Modern operating systems such as Linux-enable programs to load and unload shared libraries at run time, a process known as *dynamic loading*. Dynamic loading presents the possibility that multiple functions may be mapped to the same address at different times during a program's execution. During execution, `hpcrun` ensures that all measurements are attributed to the proper routine in such cases.

4. ANALYSIS

This section describes HPCTOOLKIT's general approach to analyzing performance measurements, correlating them with source code, and preparing them for presentation.

4.1. Correlating performance metrics with optimized code

To enable effective analysis, measurements of fully optimized programs must be correlated with important source code abstractions. Since measurements are made with reference to executables and shared libraries, for analysis it is necessary to map measurements back to the program source. To perform this translation, i.e. to associate sample-based performance measurements with the static structure of fully optimized binaries, we need a mapping between object code and its associated source code structure[¶]. HPCTOOLKIT's `hpcstruct` constructs this mapping using binary analysis; we call this process *recovering program structure*.

`hpcstruct` focuses its efforts on recovering procedures and loop nests, the most important elements of the source code structure. To recover the program structure, `hpcstruct` parses a load module's machine instructions, reconstructs its control flow graph, combines line map information with interval analysis on the control flow graph in a way that enables it to identify transformations to procedures such as inlining and to account for transformations to loops [8][‖].

Several benefits naturally accrue from this approach. First, HPCTOOLKIT can expose the structure of and assign metrics to what is actually executed, *even if source code is unavailable*. For example, `hpcstruct`'s program structure naturally reveals transformations such as loop fusion

[¶] This object to source code mapping should be contrasted with the binary's line map, which (if present) is typically fundamentally line based.

[‖] Without line map information, `hpcstruct` can still identify procedures and loops, but is not able to account for inlining or loop transformations.

DOI: 10.1002/cpe

and scalarized loops implementing Fortran 90 array notation. Similarly, it exposes calls to compiler support routines and wait loops in communication libraries of which one would otherwise be unaware. `hpcrun`'s function discovery heuristics expose distinct logical procedures within stripped binaries.

4.2. Computed metrics

Identifying application performance problems and the opportunities for tuning may require synthetic performance metrics. To identify where an algorithm is not effectively using the hardware resources, one should compute metrics that reflect the *wasted* rather than the consumed resources. For instance, when tuning a floating-point intensive scientific code, it is often less useful to know where the majority of the floating-point operations occur than where the floating-point performance is low. Knowing where the most cycles are spent doing things other than floating-point computation hints at opportunities for tuning. Such a metric can be directly computed by taking the difference between the cycle count and FLOP count divided by a target FLOPs-per-cycle value, and displaying this measure for loops and procedures. Our experiences with using multiple computed metrics such as miss ratios, instruction balance, and 'lost cycles' underscore the power of this approach.

4.3. Identifying scalability bottlenecks in parallel programs

We recently developed an MPI version of `hpcprof` which scalably analyzes, correlates, and summarizes call path profiles from large-scale executions. One novel application of HPCTOOLKIT's call path profiles is to use them to pinpoint and quantify scalability bottlenecks in SPMD parallel programs [11,12]. By combining call path profiles with program structure information, HPCTOOLKIT can use an *excess work* metric to quantify scalability losses and attribute them to the full calling context in which these losses occur.

In addition, we recently developed general techniques for effectively analyzing multithreaded applications [13,14]. Using them, HPCTOOLKIT can attribute precise measures of lock contention, parallel idleness, and parallel overhead to *user-level* calling contexts—even for multithreaded languages such as Cilk [15], which uses a work-stealing run-time system.

5. PRESENTATION

This section describes `hpcviewer` and `hpctraceview`, HPCTOOLKIT's two presentation tools. We illustrate the functionality of these tools by applying them to measurements of parallel executions of the Gyrokinetic Toroidal Code (GTC) [16]. GTC is a particle-in-cell (PIC) code for simulating turbulent transport in fusion plasma in devices such as the International Thermonuclear Experimental Reactor. GTC is a production code with 8M processor hours allocated to its executions during 2008. To briefly summarize the nature of GTC's computation, each time step repeatedly executes `charge`, `solve`, and `push` operations. In the `charge` step, it deposits the charge from each particle onto grid points nearby. Next, the `solve` step computes the electrostatic potential and field at each grid point by solving the Poisson equation on the grid. In the `push` step, the force on each particle is computed from the potential at the nearby grid points. Particles move according to the forces on them.

DOI: 10.1002/cpe

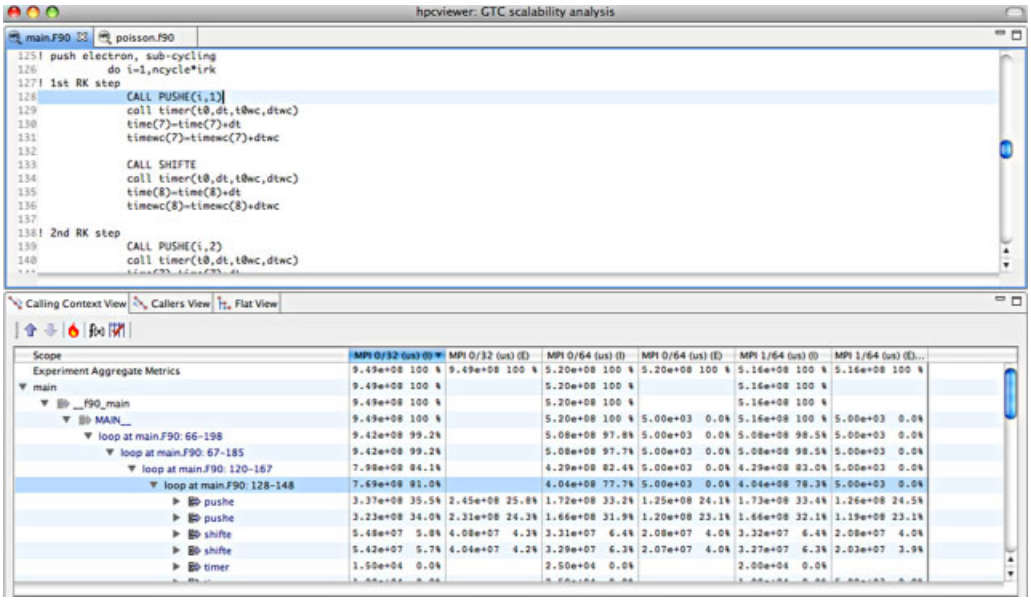


Figure 1. Using `hpcviewer` to assess the hotspots in GTC.

5.1. `hpcviewer`

HPCTOOLKIT's `hpcviewer` user interface presents performance metrics correlated to the program structure (Section 4.1) and mapped to a program's source code, if available. Figure 1 shows a snapshot of the `hpcviewer` user interface viewing data from several parallel executions of GTC. The user interface is composed of two principal panes. The top pane displays the program source code. The bottom pane associates a table of performance metrics with a static or dynamic program structure. `hpcviewer` provides three different views of the performance measurements collected using call path profiling. We briefly describe the three views and their corresponding purposes.

- *Calling context view*: This top-down view associates an execution's dynamic calling contexts with their costs. Using this view, one can readily see how much of the application's cost was incurred by a function when called from a particular context. If finer detail is of interest, one can explore how the costs incurred by a call in a particular context are divided between the callee itself and the procedures it calls. HPCTOOLKIT distinguishes calling context precisely by individual call sites; this means that if a procedure g contains calls to procedure f in different places, each call represents a separate calling context. Figure 1 shows a calling context view. This view is created by integrating the static program structure (e.g. loops) with dynamic calling contexts gathered by `hpcrun`. Loops appear explicitly in the call chains shown in Figure 1.
- *Callers view*: This bottom-up view enables one to look upward along call paths. This view is particularly useful for understanding the performance of software components or procedures that are called in more than one context. For instance, a message-passing program may call `MPI_Wait` in many different calling contexts. The cost of any particular call will depend upon

DOI: 10.1002/cpe

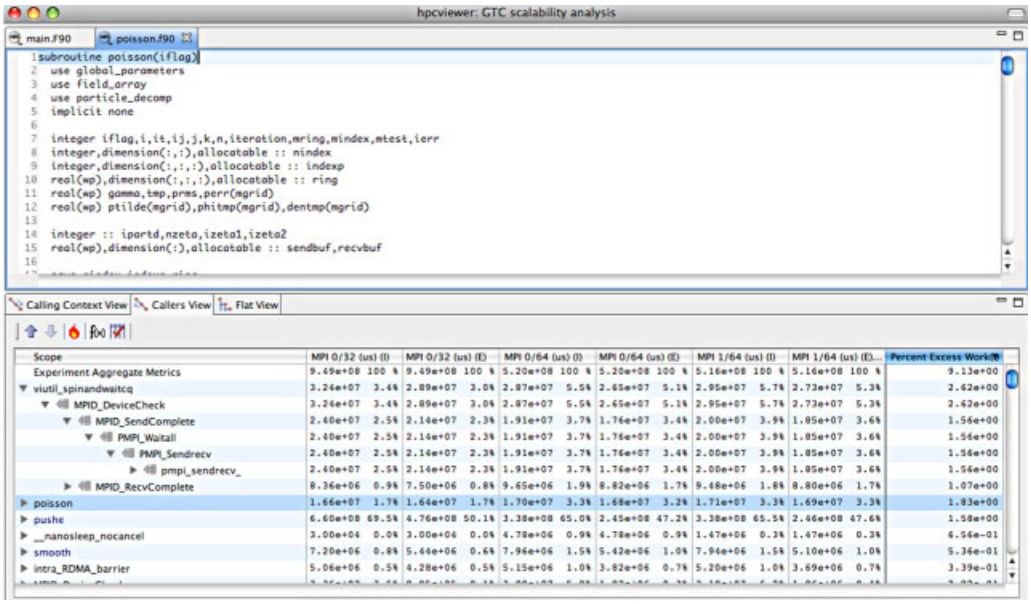


Figure 2. Using `hpcviewer` to assess the scalability of particle decomposition in GTC.

its context. Serialization or load imbalance may cause long waits in some calling contexts but not in others. Figure 2 shows a caller's view of costs for processes from two parallel runs of GTC.

- *Flat view* This view organizes the performance data according to an application's static structure. All costs incurred in any calling context by a procedure are aggregated together in the flat view. This complements the calling context view, in which the costs incurred by a particular procedure are represented separately for each call to the procedure from a different calling context.

`hpcviewer` can present an arbitrary subset of the performance metrics gathered during one or more runs, or compute the derived metrics expressed as formulae with the existing metrics as terms. For any given scope, `hpcviewer` computes both *exclusive* and *inclusive* metric values. Exclusive metrics only reflect the costs for a scope itself; inclusive metrics reflect the costs for the entire subtree rooted at that scope. Within a view, a user may order program scopes by sorting them using any performance metric. `hpcviewer` supports several convenient operations to facilitate analysis: revealing a *hot path* within the hierarchy below a scope, *flattening* out one or more levels of the static hierarchy, e.g. to facilitate comparison of costs between loops in different procedures, and *zooming* to focus on a particular scope and its children.

5.1.1. Using `hpcviewer`

In this section, we illustrate the capabilities of `hpcviewer` by using it to examine the profile data collected for GTC. The version of GTC that we studied uses a domain decomposition along the

DOI: 10.1002/cpe

toroidal dimension of a tokamak. Each toroidal domain contains one poloidal plane. One or more MPI processes can be assigned to each toroidal domain. In GTC, many of the more expensive loops are parallelized using OpenMP. Particles in each poloidal plane are randomly distributed in equal numbers to MPI processes assigned to a toroidal domain. Particles move between poloidal planes via MPI communication.

We used `hpcrun` to collect call path profiles of three parallel configurations using timer-based asynchronous sampling. All the three configurations use the same problem size and domain decomposition along the toroidal dimension; only the degree and the type of parallelism within each poloidal plane vary. The baseline configuration uses a single MPI process in each of 32 poloidal planes. The second configuration doubles the amount of parallelism by assigning a second MPI process to each plane. The third configuration uses a hybrid MPI+OpenMP approach, with two threads in each plane.

Figure 1 shows side-by-side views of the profile data collected for the MPI rank 0 process of the 32-processor run, along with the data for the MPI ranks 0 and 1—the processes for the first poloidal plane in a 64-process MPI execution. For each MPI process, we show two metrics, representing the inclusive and the exclusive wall time spent in each scope. The bottom left of Figure 1 shows the hot call path for the MPI process in the 32-process configuration. A loop nested four levels deep in the main routine accounts for 81% of the total execution time. This loop simulates the electron motion. `hpcviewer`'s ability to attribute cost to individual loops comes from information provided by `hpcstruct`'s binary analysis. The cost of simulating the electron motion is high in this simulation because electrons move through the tokamak much faster than ions and need to be simulated at a much finer time scale. From Figure 1 we notice that when we increase the parallelism by a factor of two, the contribution of the electron sub-cycle loop to the total execution time drops to approximately 78%. This is due to the less efficient scaling of other sections of the program, which we explore next.

Figure 2 presents a second snapshot of `hpcviewer` displaying a bottom-up view of the profile data shown in Figure 1. The last metric shown in this figure is a derived metric representing the percentage of excess work performed in the 64-process run relative to the 32-process run. As we doubled the amount of parallelism within each poloidal plane, the total amount of work performed by the two MPI processes for a plane was roughly 9% larger than the amount of work performed by a single MPI process in the 32-process run. Sorting the program scopes by this derived metric, as shown in Figure 2, enables us to pinpoint those routines whose execution cost has been dilated the most in absolute terms.

We notice that the routine accounting for the highest amount of excess work is `viutil_spinandwaitcq`. Expanding the calling contexts that lead to this routine reveals that this is an internal routine of the MPI library that waits for the completion of MPI operations. The second most significant routine according to our derived metric is `poisson`, a GTC routine that solves Poisson equations to compute the electrostatic potential. While this routine accounts for only 1.7% of the execution time in the baseline configuration, we see that its execution time *increases* as we double the level of parallelism in each poloidal plane. In fact, the work performed by this routine is replicated in each MPI process working on a poloidal plane. As a result, the contribution of `poisson` increases to 3.3% of the total time for the 64-process run. This routine may become a bottleneck as we increase the amount of parallelism within each poloidal plane by higher factors. On a more positive note, Figure 2 shows that routine `pushe`, which performs electron simulation

DOI: 10.1002/cpe

and accounts for 50% of the total execution time, has very good scaling. Its execution time is dilated less than that of `poisson`, causing it to be ranked lower according to the excess work metric.

This brief study of GTC shows how the measurement, analysis, attribution, and presentation capabilities of HPCTOOLKIT make it straightforward to pinpoint and quantify the reasons for subtle differences in the relative scaling of the different parallel configurations of an application.

5.2. `hpctraceview`

`hpctraceview` is a prototype visualization tool that was recently added to HPCTOOLKIT. `hpctraceview` renders space–time diagrams that show how a parallel execution unfolds over time. Figure 3 shows a screen snapshot of `hpctraceview` displaying an interval of execution for a hybrid MPI+OpenMP version of the GTC code running on 64 processors. The execution consists of 32 MPI processes with two OpenMP threads per process. Although `hpctraceview`'s visualizations on the surface seem rather similar to those by many contemporary tools, the nature of its visualizations and the data upon which they are based are rather different from that of other tools.

As we describe in more detail in Section 6, other tools for rendering execution traces of parallel programs rely on embedded program instrumentation that *synchronously* records information about the entry and exit of program procedures, communication operations, and/or program phase markers. Unlike other tools, `hpctraceview`'s traces are collected using *asynchronous* sampling. Each time line in `hpctraceview` represents a sequence of the asynchronous samples taken over the

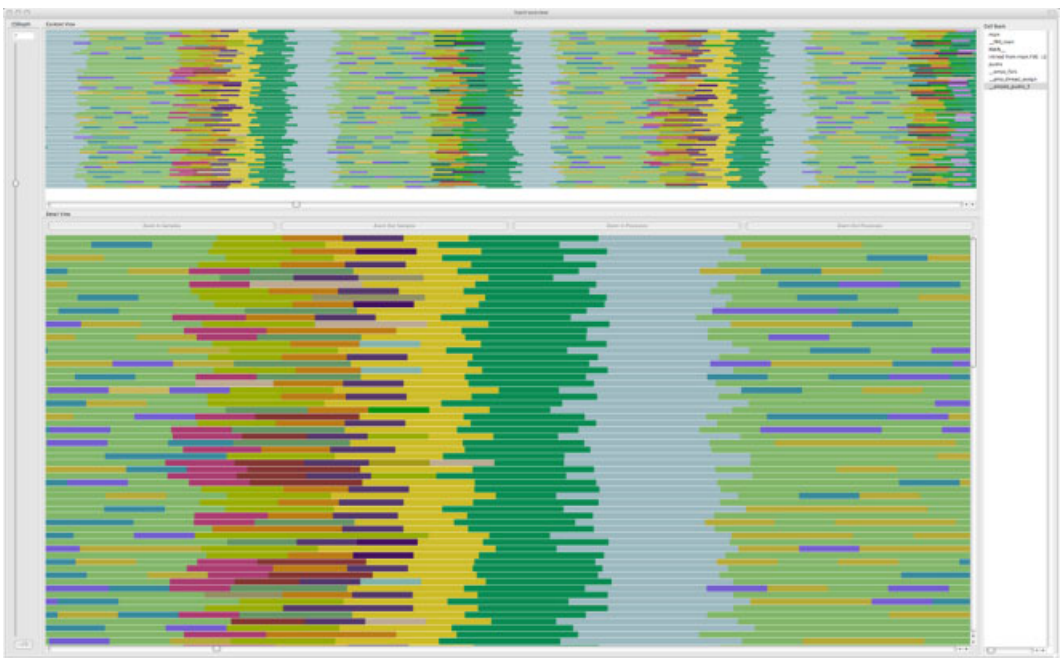


Figure 3. `hpctraceview` showing part of a execution trace for GTC.

DOI: 10.1002/cpe

life of a thread (or process). `hpctraceview`'s samples are multi-level; each sample for a thread represents the entire call stack of procedures active when the sample event occurred.

A closer look at Figure 3 reveals `hpctraceview`'s capabilities. The heart of the display is the two center panels that display a set of time lines for threads. Within each panel, the time lines for different threads are stacked from top to bottom. Even numbered threads (starting from 0) represent MPI processes; odd-numbered threads represent OpenMP slave threads. A thread's activity over time unfolds from left to right. The top center panel represents a low-resolution view of the time lines, known as the context view. Each distinct color on the time lines represents a different procedure. A casual inspection of the context view shows three complete repetitions of a 'pattern' and part of a fourth. A closer look reveals that the first and third repetitions are somewhat different in nature from the second and fourth: the aforementioned patterns contain a band of yellow bars (in B/W, this is the band of the lightest shade), whereas the latter do not. Space–time visualizations are good for spotting and understanding such temporal varying behavior. The bottom center pane shows a detailed view of the time lines in the context view. One selects a region in the context view to display it in the detail view. Within the detail view, one can scroll and zoom to adjust the content of the view. Unlike other trace visualizers, `hpctraceview`'s visualizations are hierarchical. Since each sample in each thread timeline represents a call path, we can view the thread time lines at different call path depths. To the left of the context and detail panes is a slider that can be used to set the level of the view. The space–time diagrams show colored bars representing samples at the selected stack depth; any samples at shallower depth are shown at their deepest level. Figure 3 shows the trace at a call path depth of seven. In the detailed view, one can use a pointing device to select a colored sample in the detailed view. The rightmost pane of the display shows the complete call path for the yellow bar representing the conditional copy loop for the topmost process in the context view.

6. RELATED WORK

Many performance tools focus on a particular dimension of measurement. For example, several tools use tracing [17–22] to measure how an execution unfolds over time. Tracing can provide valuable insight into phase and time-dependent behavior and is often used to detect MPI communication inefficiencies. In contrast, profiling may miss time-dependent behavior, but its measurement, analysis, and presentation strategies scale more easily to long executions. For this reason, other tools employ profiling [1,23,24]. Some tools [5,25–28], now including HPCTOOLKIT, support both profiling and tracing. Because either profiling or tracing may be the best form of measurement for a given situation, tools that support both forms have a practical advantage.

Either profiling or tracing may expose the aspects of an execution's state, such as calling context to form call path profiles or call path traces. Although other tools [5,29,30] collect calling contexts, HPCTOOLKIT is unique in supporting both call path profiling and call path tracing. In addition, our call path measurement has novel aspects that make it more accurate and impose a lower overhead than other call graph or call path profilers; a detailed comparison can be found elsewhere [8].

Tools for measuring parallel application performance are typically model dependent, such as libraries for monitoring MPI communication (e.g. [30–32]), interfaces for monitoring OpenMP

DOI: 10.1002/cpe

programs (e.g. [21,33]), or global address space languages (e.g. [34]). In contrast, HPCTOOLKIT can pinpoint contextual performance problems independent of model—and even within stripped, vendor-supplied math and communication libraries.

Although performance tools may measure the same dimensions of an execution, they may differ with respect to their measurement methodology. TAU [29], OPARI [33], and Pablo [35] among others add instrumentation to the source code during the build process. Model-dependent strategies often use instrumented libraries [21,32,36–38]. Other tools analyze unmodified application binaries by using dynamic instrumentation [5,39–41] or library preloading [4,24,27,28,42]. These different measurement approaches affect a tool's ease of use, but more importantly fundamentally affect its potential for accurate and scalable measurements. Tools that permit monitoring of unmodified executables are critical for applications with long build processes or for attaching to an existing production run. More significantly, source code instrumentation cannot measure binary-only library code, may affect compiler transformations, and incurs large overheads. Binary instrumentation may also have blind spots and incur large overheads. For example, the widely used VTune [5] call path profiler employs binary instrumentation that fails to measure functions in stripped object code and imposes enough overhead so that Intel explicitly discourages program-wide measurement. HPCTOOLKIT's call path profiler uniquely combines preloading (to monitor unmodified dynamically linked binaries), asynchronous sampling (to control overhead), and binary analysis (to assist handling of unruly object code) for measurement.

Tracing on large-scale systems is widely recognized to be costly and to produce massive trace files [32]. Consequently, many scalable performance tools manage data by collecting summaries based on synchronous monitoring (or sampling) of library calls (e.g. [30,32]) or by profiling based on asynchronous events (e.g. [1,23,24]). HPCTOOLKIT's call path tracer uses asynchronous sampling and novel techniques to manage measurement overhead and data size better than a flat tracer.

Tools for analyzing bottlenecks in parallel programs are typically *problem focused*. Paradyn [41] uses a performance problem search strategy and focused instrumentation to look for well-known causes of inefficiency. Strategies based on instrumentation of communication libraries, such as Photon and mpiP, focus only on communication performance. Vetter [43] describes an assisted learning-based system that analyzes MPI traces and automatically classifies communication inefficiencies, based on the duration of primitives, such as blocking and non-blocking send and receive. EXPERT [44] also examines communication traces for patterns that correspond to known inefficiencies. In contrast, HPCTOOLKIT's scaling analysis is *problem-independent*.

7. CONCLUSIONS AND FUTURE DIRECTIONS

Much of the focus of the HPCTOOLKIT project has been on measurement, analysis, and attribution of the performance within processor nodes. Our early work on measurement focused on 'flat' statistical sampling of the hardware performance counters that attributed costs to the instructions and loops that incurred them. As the scope of our work broadened from analysis of computation-intensive Fortran programs (whose static call graphs were often tree-like) to programs that make extensive use of multi-layered libraries, such as those for communication and math, it became important to gather and attribute information about costs to the full calling contexts in which they were incurred. HPCTOOLKIT's use of binary analysis to support both measurement (call stack

DOI: 10.1002/cpe

unwinding of unmodified optimized code) and attribution to loops and inlined functions has enabled its use on today's grand challenge applications—multi-lingual programs that leverage third-party libraries for which the source code and symbol information may not be available.

Our observation that one could use differential analysis of call path profiles to pinpoint and quantify the scalability bottlenecks led to an effective technique that can be used to pinpoint the scalability bottlenecks of all types on systems of any size, independent of the programming model. We have applied this approach to pinpoint synchronization, communication, and I/O bottlenecks on applications on large-scale distributed-memory machines. In addition, we have used this technique to pinpoint the scalability bottlenecks on multicore processors—program regions where scaling from one core to multiple cores is less than ideal.

A blind spot when our tools used profiling exclusively was understanding the program behavior that differs over time. Call path tracing and the `hpctraceview` visualizer enable us to address this issue. A benefit of our tracing approach based on asynchronous rather than synchronous sampling is that we can control the measurement overhead by reducing the sampling frequency, whereas synchronous sampling approaches have less effective options.

While we have demonstrated that our measurement and analysis techniques scale to emerging petascale systems, additional work is needed to facilitate top-down presentation of the performance data for large-scale executions. For large-scale runs, `hpcviewer` currently displays calling context metrics (min, max, mean, sum, standard deviation) that summarize the behavior of all the processes in an execution. We plan to extend `hpcviewer` to navigate from summary metrics to detailed per-thread data, which it will manipulate out-of-core. In addition, we plan to extend `hpctraceview` to visualize long executions with large numbers of processors.

ACKNOWLEDGEMENTS

HPCTOOLKIT project alumni include Nathan Froyd and Robert Fowler. Cristian Coarfa was involved in the development of scalability analysis using call path profiles.

REFERENCES

1. Mellor-Crummey JM, Fowler R, Marin G, Tallent N. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing* 2002; **23**(1):81–104.
2. Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF. Producing wrong data without doing anything obviously wrong! *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM: New York, NY, U.S.A., 2009; 265–276.
3. Graham SL, Kessler PB, McKusick MK. Gprof: A call graph execution profiler. *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. ACM Press: New York, NY, U.S.A., 1982; 120–126.
4. Froyd N, Mellor-Crummey JM, Fowler R. Low-overhead call path profiling of unmodified, optimized code. *Proceedings of the 19th Annual International Conference on Supercomputing*. ACM Press: New York, NY, U.S.A., 2005; 81–90.
5. Intel Corporation. Intel VTune performance analyzer. Available at: http://software.intel.com/en-us/intel-vtune [2 December 2009].
6. Intel Corporation. Intel Performance Tuning Utility. Available at: http://software.intel.com/en-us/articles/intel-performance-tuning-utility [2 December 2009].
7. Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press: New York, NY, U.S.A., 2005; 190–200.

DOI: 10.1002/cpe

8. Tallent NR, Mellor-Crummey JM, Fagan MW. Binary analysis for measurement and attribution of program performance. *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., 2009; 441–452.
9. Shende S, Malony A, Morris A. *Optimization of Instrumentation in Parallel Performance Evaluation Tools* (*Lecture Notes in Computer Science*, vol. 4699). Springer: Berlin, 2008; 440–449.
10. Ammons G, Ball T, Larus JR. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., 1997; 85–96.
11. Coarfa C, Mellor-Crummey JM, Froyd N, Dotsenko Y. Scalability analysis of SPMD codes using expectations. *ICS '07*: *Proceedings of the 21st Annual International Conference on Supercomputing*. ACM: New York, NY, U.S.A., 2007; 13–22.
12. Tallent NR, Mellor-Crummey JM, Adhianto L, Fagan MW, Krentel M. Diagnosing performance bottlenecks in emerging petascale applications. *Proceedings of the Conference on High Performance Computing Networking*, *Storage and Analysis. SC '09*. ACM: New York, NY, 2009; 1–11. DOI: http://doi.acm.org/10.1145/1654059.1654111.
13. Tallent NR, Mellor-Crummey JM. Effective performance measurement and analysis of multithreaded applications. *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM: New York, NY, U.S.A., 2009; 229–240.
14. Tallent NR, Mellor-Crummey JM, Porterfield A. Analyzing lock contention in multithreaded applications. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, 2010.
15. Frigo M, Leiserson CE, Randall KH. The implementation of the Cilk-5 multithreaded language. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Que., Canada, 1998; 212–223.
16. Lin Z, Hahm TS, Lee WW, Tang WM, White RB. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science* 1998; **281**(5384):1835–1837.
17. Nagel WE, Arnold A, Weber M, Hoppe HC, Solchenbach K. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer* 1996; **12**(1):69–80.
18. Gu W, Eisenhauer G, Schwan K, Vetter J. Falcon: On-line monitoring for steering parallel programs. *Concurrency*: *Practice and Experience* 1998; **10**(9):699–736.
19. Zaki O, Lusk E, Gropp W, Swider D. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications* 1999; **13**(2):277–288.
20. Worley PH. MPICL: A port of the PICL tracing logic to MPI. Available at: http://www.csm.ornl.gov/picl [2 December 2009].
21. Caubet J, Gimenez J, Labarta J, Rose LD, Vetter JS. A dynamic tracing mechanism for performance analysis of OpenMP applications. *Proceedings of the International Workshop on OpenMP Applications and Tools*. Springer: London, U.K., 2001; 53–67.
22. Wolf F, Mohr B. EPILOG binary trace-data format. *Technical Report FZJ-ZAM-IB-2004-06*, Forschungszentrum Julich, May 2004.
23. Anderson TE, Lazowska ED. Quartz: A tool for tuning parallel program performance. *SIGMETRICS Performance Evaluation Review* 1990; **18**(1):115–125.
24. Cortesi D, Fier J, Wilson J, Boney J. Origin 2000 and Onyx2 performance tuning and optimization guide. *Technical Report 007-3430-003*, Silicon Graphics, Inc., 2001.
25. Fürlinger K, Gerndt M, Dongarra J. On using incremental profiling for the performance analysis of shared memory parallel applications. *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Rennes, France, 2007; 62–71.
26. Morris A, Spear W, Malony AD, Shende S. Observing performance dynamics using parallel profile snapshots. *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*. Springer: Berlin, Heidelberg, 2008; 162–171.
27. Silicon Graphics, Inc. (SGI). SpeedShop User's Guide. *Technical Report 007-3311-011*, SGI, 2003.
28. Krell Institute. Open SpeedShop for Linux. Available at: http://www.openspeedshop.org.
29. Shende SS, Malony AD. The TAU parallel performance system. *International Journal of High Performance Computing Applications* 2006; **20**(2):287–311.
30. Vetter JS, McCracken MO. Statistical scalability analysis of communication operations in distributed applications. *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, 2001.
31. Wu CE, Bolmarcich A, Snir M, Wootton D, Parpia F, Chan A, Lusk E, Gropp W. From trace generation to visualization: A performance framework for distributed parallel systems. *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society: Washington, DC, U.S.A., 2000.
32. Vetter J. Dynamic statistical profiling of communication activity in distributed applications. *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM Press: New York, NY, U.S.A., 2002; 240–250.

33. Mohr B, Malony AD, Shende S, Wolf F. Design and prototype of a performance tool interface for OpenMP. *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, October 2001.
34. Su H-H, Bonachea D, Leko A, Sherburne H Billingsley III M. George AD. GASP! A standardized performance analysis tool interface for global address space programming models. *Technical Report LBNL-61659*, Lawrence Berkeley National Laboratory, 2006.
35. Reed DA, Aydt RA, Noe RJ, Roth PC, Shields KA, Schwartz BW, Tavera LF. Scalable performance analysis: The Pablo performance analysis environment. *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society: Silver Spring, MD, 1993; 104–113.
36. Mohr B, Malony AD, Hoppe H-C, Schlimbach F, Haab G, Hoeflinger J, Shah S. A performance monitoring interface for OpenMP. *Proceedings of the Fourth European Workshop on OpenMP*, Rome, Italy, 2002.
37. Fürlinger K, Gerndt M. ompP: A profiling tool for OpenMP. *Proceedings of the First and Second International Workshops on OpenMP* (*Lecture Notes in Computer Science*, vol. 4315), Eugene, OR, U.S.A., 2005; 12–23.
38. Schulz M, de Supinski RB. P^NMPI tools: A whole lot greater than the sum of their parts. *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. ACM: New York, NY, U.S.A., 2007; 1–10.
39. Buck B, Hollingsworth JK. An API for runtime code patching. *The International Journal of High Performance Computing Applications* 2000; **14**(4):317–329.
40. DeRose L, Ted Hoover J, Hollingsworth JK. The dynamic probe class library—An infrastructure for developing instrumentation for performance tools. *Proceedings of the International Parallel and Distributed Processing Symposium*, San Francisco, CA, U.S.A., April 2001.
41. Miller BP, Callaghan MD, Cargille JM, Hollingsworth JK, Irvin RB, Karavanic KL, Kunchithapadam K, Newhall T. The Paradyn parallel performance measurement tool. *IEEE Computer* 1995; **28**(11):37–46.
42. Mucci PJ. PapiEx—Execute arbitrary application and measure hardware performance counters with PAPI. Available at: http://icl.cs.utk.edu/~mucci/papiex [2 December 2009].
43. Vetter J. Performance analysis of distributed applications using automatic classification of communication inefficiencies. *International Conference on Supercomputing*, Santa Fe, NM, U.S.A., 2000; 245–254.
44. Wolf F, Mohr B, Dongarra J, Moore S. Efficient pattern search in large traces through successive refinement. *Proceedings of the European Conference on Parallel Computing*, Pisa, Italy, August 2004.

DOI: 10.1002/cpe