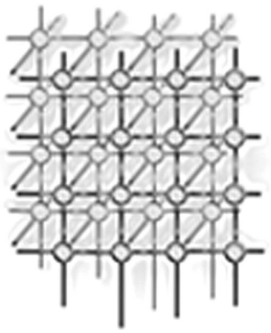
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

HPCTOOLKIT: Tools for performance analysis of optimized parallel programs‡



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

Rice University, Dept. of Computer Science
P.O. Box 1892, Houston, TX 77251-1892, USA

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.

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 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 latency and improving 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

*Correspondence to: Rice University, Dept. of Computer Science, P.O. Box 1892, Houston, TX 77251-1892, USA

†E-mail: johnmc@rice.edu

‡WWW: `http://hpctoolkit.org`

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

Received 22 Aug 2008
Revised 31 Oct 2009

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 the largest 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 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 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. 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 in more detail. 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 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 identified several of these principles in earlier work [19], it is helpful to revisit them as they continually stimulate our ideas for revision and enhancement.

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 application performance through a variety of mechanisms [25]. The most common problem is overhead, which distorts measurements. The classic tool `gprof` [12], which uses compiler-inserted instrumentation, induced an average overhead of over 100% on the SPEC 2000 integer benchmarks [9]. Intel's VTune [15], which uses static binary instrumentation, claims an average overhead of a factor of eight. Intel's Performance Tuning Utility (PTU) [14] includes a call graph profiler based on Pin's

dynamic binary instrumentation [18]; we found that it yielded an average overhead of over 400% on the SPEC 2006 integer benchmarks [34].

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 application performance by interfering with inlining and template optimization. To avoid these effects, tools such as TAU intentionally refrain from instrumenting certain procedures [29]. 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 source code is not available. To avoid systematic error, one must measure costs for 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 analysis.

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 there are often 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 measure of not where resources are consumed, but where they are consumed *inefficiently*. For this purpose, derived measures such as the difference between peak and 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 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 quickly locate bottlenecks without the need to wade through irrelevant details. HPCTOOLKIT's user interface supports hierarchical presentation of performance data according to both static and dynamic contexts, along with ranking and sorting based on metrics.

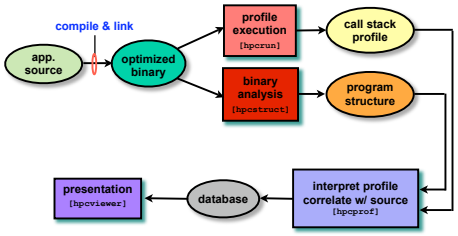
Hierarchical aggregation is vital. The amount of instruction-level parallelism in processor cores can make it difficult or expensive for hardware counters to precisely attribute particular events to specific instructions. However, even if fine-grain attribution of events is flawed, total event counts 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 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 accompanying figure. The workflow is organized around four principal capabilities:

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



HPCTOOLKIT workflow.

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.

[†]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.

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 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 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 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 rest 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) [1]. In a calling context tree, 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 leaf 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 [34]. 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 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. 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 [34].

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 source code structure. To recover program structure, `hpcstruct` parses a load module's machine instructions, reconstructs a 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 account for transformations to loops [34].[§]

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 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 performance problems and opportunities for tuning may require synthetic performance metrics. To identify where an algorithm is not effectively using hardware resources, one should compute a metric that reflects *wasted* rather than 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 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 [5, 36]. Combining call path profiles with program structure information, HPCTOOLKIT can

[‡]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.

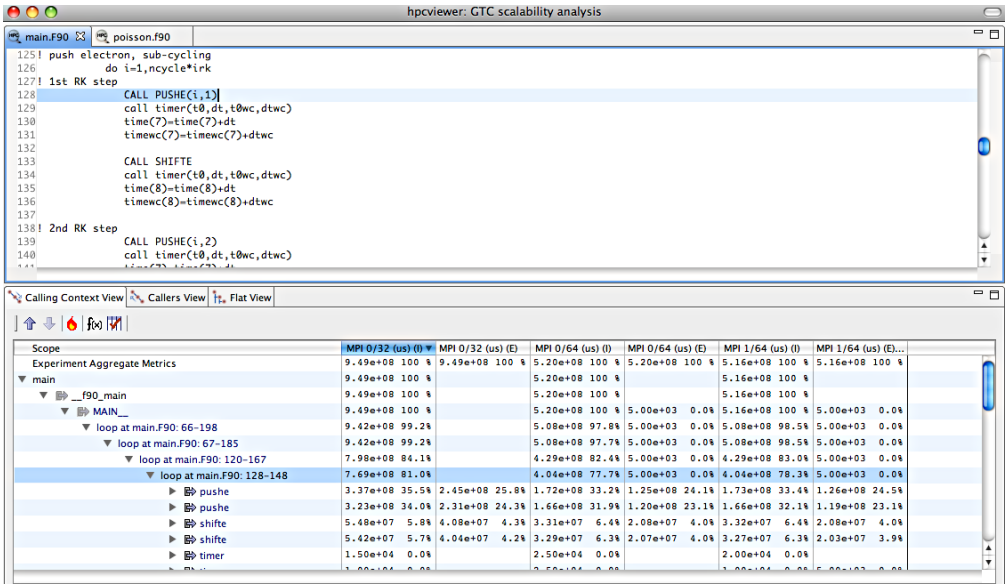


Figure 1. Using `hpcviewer` to assess the hotspots in GTC.

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 techniques general for effectively analyzing multithreaded applications [33, 35]. 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 [8], 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) [17]. GTC is a particle-in-cell (PIC) code for simulating turbulent transport in fusion plasma in devices such as the International Thermonuclear Experimental Reactor (ITER). 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 nearby grid points. Particles move according to the forces on them.

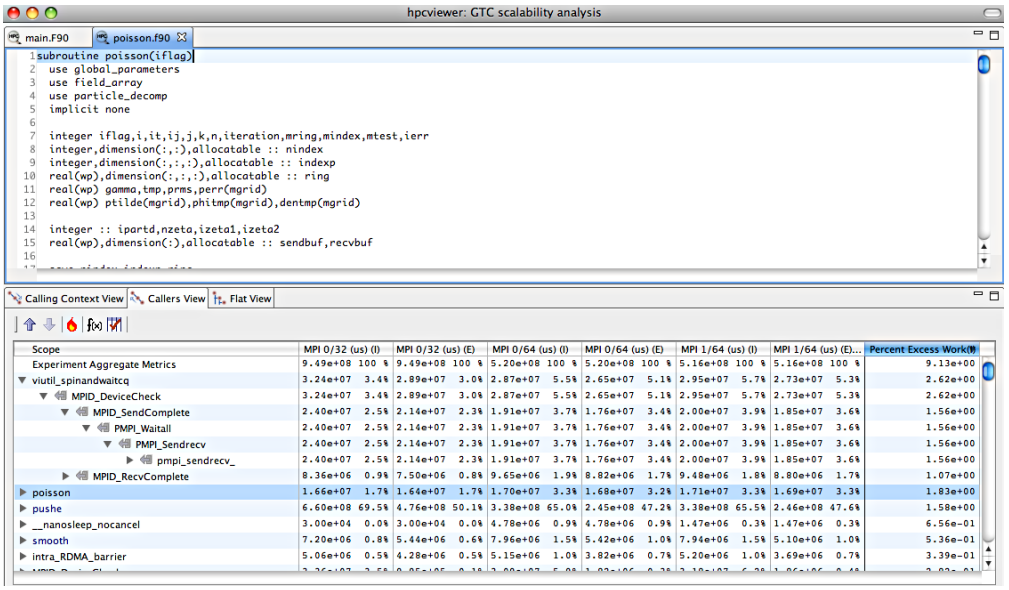


Figure 2. Using `hpcviewer` to assess the scalability of particle decomposition in GTC.

5.1 `hpcviewer`

HPCTOOLKIT's `hpcviewer` user interface presents performance metrics correlated to 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 program source code. The bottom pane associates a table of performance metrics with static or dynamic program structure. `hpcviewer` provides three different views of 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 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 its context. Serialization or load imbalance may cause long waits in some calling contexts but not others. Figure 2 shows a caller's view of costs for processes from two parallel runs of GTC.

- **Flat view**. This view organizes 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 collection of performance metrics gathered during one or more runs, or compute derived metrics expressed as formulae with existing metrics as terms. For any given scope, `hpcviewer` computes both *exclusive* and *inclusive* metric values. Exclusive metrics only reflect costs for a scope itself; inclusive metrics reflect 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 profile data collected for GTC. The version of GTC that we studied uses a domain decomposition along the 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 number 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 three configurations use the same problem size and domain decomposition along the toroidal dimension; only the degree and type of parallelism within each poloidal plane varies. 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 profile data collected for the MPI rank 0 process of the 32-processor run, along with 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 electron motion. `hpcviewer`'s ability to attribute cost to individual loops comes from information provided by `hpcstruct`'s binary analysis. The cost of simulating 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 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 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 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 is rather different than 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 asynchronous samples taken over the 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, time lines for different threads are stacked 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 left to right. The top center panel represents a low-resolution view of the time lines, known as the context

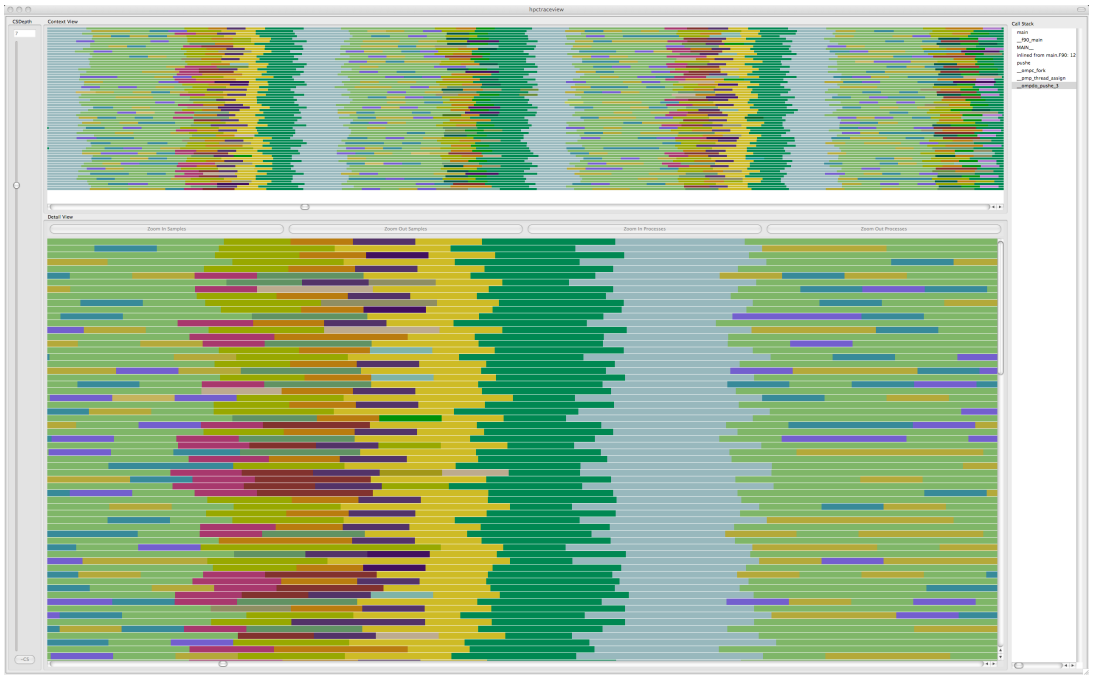


Figure 3. `hpctraceview` showing part of a execution trace for GTC.

view. Each distinct color on the time lines represents a different procedure. 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 than the second and fourth: the aforementioned patterns contain a band of yellow bars, 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 timelines 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 timelines 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 [4, 13, 26, 40, 42, 44] 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 [2, 6, 19]. Some tools [11, 15, 16, 23, 31], 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 aspects of an execution's state such as calling context to form call path profiles or call path traces. Although other tools [15, 30, 39] 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 lower overhead than other call graph or call path profilers; a detailed comparison can be found elsewhere [34].

Tools for measuring parallel application performance are typically model dependent, such as libraries for monitoring MPI communication (*e.g.*, [38, 39, 43]), interfaces for monitoring OpenMP programs (*e.g.*, [4, 22]), or global address space languages (*e.g.*, [32]). 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 [30], OPARI [22], and Pablo [27] among others add instrumentation to source code during the build process. Model-dependent strategies often use instrumented libraries [4, 10, 21, 28, 38]. Other tools analyze unmodified application binaries by using dynamic instrumentation [3, 7, 15, 20] or library preloading [6, 9, 16, 24, 31]. 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 [15] call path profiler employs binary instrumentation that fails to measure functions in stripped object code and imposes enough overhead 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 [38]. Consequently, many scalable performance tools manage data by collecting summaries based on synchronous monitoring (or sampling) of library calls (*e.g.*, [38, 39]) or by profiling based on asynchronous events (*e.g.*, [2, 6, 19]). 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 [20] 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 [37] 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 [41] 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 performance within processor nodes. Our early work on measurement focused on "flat" statistical sampling of 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 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 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 scalability bottlenecks led to an effective technique that can be used to pinpoint 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 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 program behavior that differs over time. Call path tracing and the `hpctraceview` visualizer enables us to address this issue. A benefit of our tracing approach based on asynchronous rather than synchronous sampling is that we can control measurement overhead by reducing 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 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 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. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, NY, NY, USA, 1997. ACM.
2. T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. *SIGMETRICS Perform. Eval. Rev.*, 18(1):115–125, 1990.
3. B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
4. J. Caubet, J. Gimenez, J. Labarta, L. D. Rose, and J. S. Vetter. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *Proc. of the Intl. Workshop on OpenMP Appl. and Tools*, pages 53–67, London, UK, 2001. Springer-Verlag.
5. C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of SPMD codes using expectations. In *ICS '07: Proc. of the 21st annual International Conference on Supercomputing*, pages 13–22, NY, NY, USA, 2007. ACM.
6. D. Cortesi, J. Fier, J. Wilson, and J. Boney. Origin 2000 and Onyx2 performance tuning and optimization guide. Technical Report 007-3430-003, Silicon Graphics, Inc., 2001.
7. L. DeRose, J. Ted Hoover, and J. K. Hollingsworth. The dynamic probe class library - an infrastructure for developing instrumentation for performance tools. *Proc. of the International Parallel and Distributed Processing Symposium*, April 2001.
8. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998.
9. N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. of the 19th annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.
10. K. Fürlinger and M. Gerndt. ompP: A profiling tool for OpenMP. In *Proc. of the First and Second International Workshops on OpenMP*, pages 15–23, Eugene, Oregon, USA, May 2005. LNCS 4315.
11. K. Fürlinger, M. Gerndt, and J. Dongarra. On using incremental profiling for the performance analysis of shared memory parallel applications. In *Proc. of the 13th International Euro-Par conference on Parallel Processing*, pages 62–71, 2007.
12. S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *Proc. of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.
13. W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.
14. Intel Corporation. Intel Performance Tuning Utility. `http://software.intel.com/en-us/articles/intel-performance-tuning-utility`.
15. Intel Corporation. Intel VTune performance analyzer. `http://www.intel.com/software/products/vtune`.
16. Krell Institute. Open SpeedShop for Linux. `http://www.openspeedshop.org`.
17. Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, and R. B. White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, 281(5384):1835–1837, September 1998.
18. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 ACM SIGPLAN conference on programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.
19. J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.
20. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.
21. B. Mohr, A. D. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah. A performance monitoring interface for OpenMP. In *Proceedings of the Fourth European Workshop on OpenMP*, Rome, Italy, 2002.
22. B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2001.
23. A. Morris, W. Spear, A. D. Malony, and S. Shende. Observing performance dynamics using parallel profile snapshots. In *Proc. of the 14th International Euro-Par conference on Parallel Processing*, pages 162–171, Berlin, Heidelberg, 2008. Springer-Verlag.
24. P. J. Mucci. PapiEx - execute arbitrary application and measure hardware performance counters with PAPI. `http://icl.cs.utk.edu/~mucci/papiex`.
25. T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. of the 14th international conference on Architectural support for programming languages and operating systems*, pages 265–276, New York, NY, USA, 2009. ACM.

26. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
27. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proc. of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.
28. M. Schulz and B. R. de Supinski. P^NMPI tools: a whole lot greater than the sum of their parts. In *Proc. of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.
29. S. Shende, A. Malony, and A. Morris. *Optimization of Instrumentation in Parallel Performance Evaluation Tools*, volume 4699 of *LNCS*, pages 440–449. Springer, 2008.
30. S. S. Shende and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.
31. Silicon Graphics, Inc. (SGI). SpeedShop User's Guide. Technical Report 007-3311-011, SGI, 2003.
32. H.-H. Su, D. Bonachea, A. Leko, H. Sherburne, M. B. III, and A. D. George. GASP! a standardized performance analysis tool interface for global address space programming models. Technical Report LBNL-61659, Lawrence Berkeley National Laboratory, 2006.
33. N. R. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–240, New York, NY, USA, 2009. ACM.
34. N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 441–452, New York, NY, USA, 2009. ACM.
35. N. R. Tallent, J. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.
36. N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel. Diagnosing performance bottlenecks in emerging petascale applications. In *Proc. of the 2009 ACM/IEEE Conference on Supercomputing*, 2009.
37. J. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *International Conference on Supercomputing*, pages 245–254, 2000.
38. J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 240–250, NY, NY, USA, 2002. ACM Press.
39. J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, 2001.
40. F. Wolf and B. Mohr. EPILOG binary trace-data format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Julich, May 2004.
41. F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient pattern search in large traces through successive refinement. In *Proc. of the European Conference on Parallel Computing*, Pisa, Italy, Aug. 2004.
42. P. H. Worley. MPICL: a port of the PICL tracing logic to MPI. `http://www.epm.ornl.gov/picl`.
43. C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2000. IEEE Computer Society.
44. O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.