HPCToolkit: Performance Measurement and Analysis for Supercomputers with Node-level Parallelism

Laksono Adhianto, Michael Fagan, Mark Krentel,
Gabriel Marin, John Mellor-Crummey, Nathan Tallent
Dept. of Computer Science, Rice University, Houston, TX, USA
{laksono, mfagan, krentel, mgabi, johnmc, tallent}@cs.rice.edu

Abstract

Today's largest supercomputers consist of tens of thousands of nodes equipped with one or more multi-core microprocessors. A challenge for performance tools is that bottlenecks in programs executing on these systems may arise from a myriad of causes. To address this problem, Rice University is developing HPCTOOLKIT *- an integrated suite of tools that supports sampling-based measurement, analysis, attribution, and presentation of application performance for fully-optimized parallel programs. This paper provides a brief overview of performance analysis challenges on supercomputers with node-level parallelism, describes how HPCToolkit supports a variety of performance analysis strategies that can pinpoint and quantify impediments to scalable high performance in parallel applications both within and across nodes, and outlines some remaining challenges ahead.*

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 multi-core microprocessors. Often individual 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.

As the complexity of HPC systems has grown over the years, the complexity of applications has grown as well. Achieving high efficiency with sophisticated applications on HPC systems is imperative for tackling "grand challenge" problems. As a result, there is an urgent need for effective and scalable tools that can pinpoint performance and scalability bottlenecks in complex applications running on HPC platforms. A challenge for tools is that bottlenecks may arise from a myriad of causes both within and across nodes. Ideally, a performance analysis tool will enable one to pinpoint where in the program bottlenecks occur and identify their underlying causes.

To address this problem, at Rice University we have been developing HPCTOOLKIT—a suite of tools to support measurement and analysis of application performance on scalable parallel systems. This short paper provides an overview of HPCTOOLKIT and describes ways we use its capabilities to identify performance bottlenecks on scalable parallel systems built from multicore microprocessors. Section 2 provides a brief overview of HPCTOOLKIT's capabilities for measurement, attribution, and presentation of application performance data. Section 3 describes how we employ HPCTOOLKIT's capabilities to analyze application performance and pinpoint bottlenecks both within and across nodes in scalable parallel systems. Section 4 identifies open issues that are the subject of future work.

2 Overview of HPCToolkit

HPCTOOLKIT [1, 5] 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. We briefly describe the nature of these capabilities.

2.1 Measurement

Without accurate measurement, performance analysis is useless. As a result, a principal focus of work on HPCTOOLKIT has been the design and implementation of techniques for providing accurate fine-grain measurements of production applications running at scale. For tools to be useful on production applications at scale, large measurement overhead is unacceptable. For measurements to be accurate, performance tools must avoid introducing measurement error, *e.g.*, by disproportionally dilating the costs of short procedures. To address these challenges, HPCTOOLKIT avoids instrumentation and favors the use of *statistical sampling* to measure and attribute performance metrics. During a program execution, sample events are triggered by periodic interrupts induced by an interval timer or overflow of hardware performance counters. One can sample metrics that reflect work (*e.g.*, instructions, floating-point operations), consumption of resources (*e.g.*, cycles, memory bus transactions), or inefficiency (*e.g.*, stall cycles).

For all but the most trivially structured programs, it is important to associate the costs incurred by each procedure with the contexts in which the procedure is called. Knowing the context in which costs are incurred is essential for understanding performance. This is particularly important for code based on application frameworks and libraries. For instance, 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 how they are used in a particular 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. To provide insight into an application's dynamic behavior, HPCTOOLKIT also offers an option to collect traces of call path samples to provide insight into how an application's behavior unfolds over time.

For high-level multithreaded parallel programming models such as OpenMP and Cilk [3], using call path profiling to associate costs with the context in which they are incurred is not as simple as it sounds. For example, Cilk's work stealing run-time system causes calling contexts to become separated in space and time as procedure frames are stolen and migrate between threads. As a result, a standard call path profile of a Cilk program during execution will show fragments of call paths mapped to each of the threads in the run-time system's thread pool, a result that is at best cumbersome and at worst incomprehensible. For effective performance analysis of multithreaded programming models with sophisticated run-time systems, it is important to bridge the gap between the abstractions of the user's program and their realization at run time. We have developed *logical call path profiles*, a generalization of call path profiles, to enable one to measure and correlate execution behavior at different levels of abstraction [8].

2.2 Attribution

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 associate sample-based performance measurements with the static structure of fully-optimized executables, we need a mapping between object code and its associated source code structure.[1] 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 [7].[2]

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 that arise from compilation of 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.

2.3 Presentation

It is important that performance measurements generate insight that is *actionable*. For this reason, HPCTOOLKIT's `hpcviewer` user interface presents performance metrics in the context of a program's source. `hpcviewer` can present an arbitrary collection of performance metrics gathered during one or more runs, or compute new derived metrics. Having multiple views of performance data facilitates analysis. `hpcviewer` provides three different views.

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.

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

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

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, the cost of any call to `MPI_Wait` will certainly depend upon the context in which it is called. The caller's view enables one to see what proportion of the total cost attributed to a procedure was incurred in each context that the procedure was called.

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. The flat view shows the detailed breakdown of a procedure's cost among its loops, source lines, and inlined code.

`hpcviewer` supports several operations to facilitate rapid analysis: computing *user-defined derived metrics* to facilitate analysis,[3] *rank ordering* program scopes within a view by sorting them according to any metric, revealing a *hot path* within the hierarchy below a scope, and *flattening* one or more levels of the static hierarchy, *e.g.*, to facilitate a comparison of costs between loops in different procedures.

3 Analysis Strategies

The execution behavior of programs on modern parallel systems is often extraordinarily complex. It is possible to collect a wealth of information about a program execution; however, measurements themselves are not equal to insight. To understand program performance and identify bottlenecks, one must have effective analysis strategies. Here we describe a few strategies that we employ for interpreting measurement data collected with HPCToolkit.

Derived metrics. Identifying performance problems and quantifying the potential impact of tuning typically requires calculating performance metrics that cannot be measured directly. 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 to identify and quantify room for improvement. *Floating point waste* can be computed for each program scope (*e.g.*, loop, procedure, call chain) by multiplying the number of cycles spent in the scope and subtracting from that result the actual number of floating point operations performed in the given scope. Using this metric to rank order scopes highlights the code regions where an application has the greatest opportunities for improving floating point performance. Derived metrics can also be used to calculate rate limiting factors for each program scope—*e.g.*, metrics to determine whether the scope is compute, latency, or bandwidth bound—so that appropriate tuning strategies can be pursued.

[3]We describe the utility of derived metrics in the next section.

Differential profiling. McKenney [4] describes *differential profiling*—a strategy for analysis of two or more executions by mathematically combining corresponding buckets of different execution profiles. Different combining functions are useful for different situations. While McKenney applied differential profiling to flat profiles, we apply it to call path profiles. Next, we describe how we use differential profiling pinpointing scalability bottlenecks.

Scalability analysis. To pinpoint and quantify scalability bottlenecks *in context* both within and across nodes in a cluster of multicore processors, we compute a derived metric—the *fraction of excess work*—by scaling and differencing call path profiles from a pair of executions [2].

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

Using HPCToolkit, we collect call path profiles on each of p and q processors to measure the cost associated with each calling context in each execution. HPCToolkit's `hpcrun` profiler uses a data structure called a *calling context tree* (CCT) to record a call path profile. Each node in a CCT is identified by a code address. In a CCT, the path from any node to the root represents a calling context. Each node has a weight $w \geq 0$ indicating the exclusive cost attributed to the path from that node to the root. Given a pair of CCTs, one collected on p processors and another collected on q processors, with perfect weak scaling, the cost attributed to all pairs of corresponding CCT nodes[4] should be identical. Any additional cost for a CCT node on q processors when compared to its counterpart in a CCT for an an execution on p processors represents *excess work*. The fraction of excess work, *i.e.*, the amount of excess work in a calling context in a q process execution divided by the total amount of work in a q process execution represents the scalability loss attributed to that calling context. By scaling the costs attributed in a CCT before differencing them to compute excess work, one can also use this strategy to pinpoint and quantify strong scalability losses [2]. This analysis strategy is independent of the programming model and bottleneck cause.

[4]A node i in one CCT corresponds to a node j in a different CCT if the sequence of nodes along the path from i to root and the sequence of nodes from j to root are labeled with the same sequence of code addresses.

Above, we described applying our scalability analysis technique *across* nodes in a cluster. We have also applied this technique to pinpoint scaling bottlenecks within multicore nodes. For instance, one might want to understand how performance scales when using all of the cores in a node with multicore processors instead of just a single core. This can be accomplished by measuring an execution on a single core, measuring an execution on all cores, and then comparing the costs incurred by a core in each of the executions using the strategy described above for analysis of weak scaling. We have used this strategy to pinpoint and quantify scaling bottlenecks on multicore nodes at the loop level [6]. Measurements of L2 cache misses showed that contention in the memory hierarchy was the problem.

Bottleneck analysis for multithreaded codes. We recently developed general techniques for effectively analyzing multithreaded applications [8]. Using them, HPCTOOLKIT can attribute precise measures of parallel work, idleness (which might be due to load imbalance or serialization), and parallelization overhead to *user-level* calling contexts. We have applied this strategy to the Cilk multithreaded language [3], which uses a work-stealing run-time system. Similarly, it could be applied to other multithreaded programming models such as PThreads and OpenMP.

To pinpoint and quantify insufficient parallelism in executions of multithreaded programs, we measure two quantities at sample points: the number of threads performing useful work ($\mathcal{W}$) and those that are idle ($\mathcal{I}$). If a sample event occurs in a thread that is idle, we ignore it. When a sample event occurs in a thread that is actively working, the thread records one sample in a metric representing the thread's work and in a second metric, the thread records a fractional sample $\mathcal{I}/\mathcal{W}$ to charge it a proportional share of its responsibility for not keeping the idle processors busy at that moment at that point in the program. From this, we can compute a metric representing the loss of parallelism and inefficiency in each program context.

To pinpoint parallelization overhead in compiler-generated multithreaded code, we observe that a compiler for a multi-threaded programming model, such as OpenMP, can tag statements in its generated code to indicate which are associated with parallelization overhead. Using binary analysis, we can recover information recorded by the compiler and identify instructions associated with these source lines and attribute any samples associated with these lines to parallelization overhead.

Values for insufficient parallelism and parallel overhead can be attributed to the loops, procedures, and calling contexts of a program. When they are combined with derived metrics measuring instruction mix, memory bandwidth, memory latency and pipeline stalls, we can directly assess the effectiveness of a parallelization and provide guidance for how to improve it.

4 Open Issues and Future Work

To date, the HPCTOOLKIT project has focused on techniques for measurement, analysis, attribution, and presentation of performance information for node programs. Measurement capabilities could be improved with support to record system-wide information and information from the communication fabric. At present, analysis of parallel programs is largely performed by analyzing the performance of one or more processes from an individual run or comparing the performance of selected processes from different runs. To improve one's perspective, HPCTOOLKIT needs to provide statistical characterizations of performance across all processes in a parallel run. Major challenges for dealing with executions on systems with tens to hundreds of thousands of cores include parallel analysis of massive performance data, user interface support for presentation of performance data too large to fit in memory, and support for diagnosing causes of bottlenecks. Also, new techniques are needed for analysis of coupled codes.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. Technical Report TR08-06, Dept. of Computer Science, Rice University, August 2008.

[2] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of SPMD codes using expectations. In *Proc. of the 21st Annual International Conference on Supercomputing*, pages 13–22, Seattle, Washington, 2007. ACM.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[4] P. E. McKenney. Differential profiling. *Software: Practice and Experience*, 29(3):219–234, 1999.

[5] Rice University. HPCToolkit performance tools. `http://hpctoolkit.org`.

[6] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel. HPCToolkit: performance tools for scientific computing. *Journal of Physics: Conference Series*, 125:012088 (5pp), 2008.

[7] N. R. Tallent. Binary analysis for attribution and interpretation of performance measurements on fully-optimized code. M.S. thesis, Dept. of Computer Science, Rice University, May 2007.

[8] N. R. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. Technical Report TR08-05, Dept. of Computer Science, Rice University, 2008.