HPCToolkit: Performance Tools for Scientific Computing

Nathan Tallent, John Mellor-Crummey, Laksono Adhianto, Michael Fagan, Mark Krentel

Department of Computer Science, Rice University, Houston, TX, USA

E-mail: {tallent, johnmc, laksono, mfagan, krentel}@cs.rice.edu

Abstract. As part of the US Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program, science teams are tackling problems that require simulation and modeling on petascale computers. As part of activities associated with the SciDAC Center for Scalable Application Development Software (CScADS) and the Performance Engineering Research Institute (PERI), Rice University is building software tools for performance analysis of scientific applications on the leadership-class platforms. In this poster abstract, we briefly describe the HPCTOOLKIT performance tools and how they can be used to pinpoint bottlenecks in SPMD and multi-threaded parallel codes. We demonstrate HPCTOOLKIT's utility by applying it to two SciDAC applications: the S3D code for simulation of turbulent combustion and the MFDn code for *ab initio* calculations of microscopic structure of nuclei.

1. Introduction

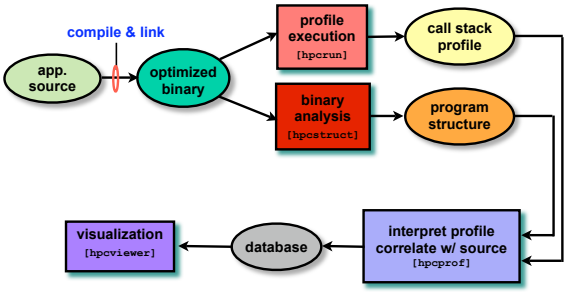
Harnessing the power of emerging petascale computational platforms to aid scientific discovery poses a grand challenge for computer science. Programming petascale systems requires both exploiting coarse-grain parallelism accross nodes and making effective use of individual microprocessor-based nodes. Performance tools that help pinpoint bottlenecks and quantify opportunities for improvement are essential for tuning codes to use these platforms efficiently.

As part of activities associated with the SciDAC Center for Scalable Application Development Software (CScADS) and the Performance Engineering Research Institute (PERI), Rice University is developing HPCTOOLKIT [1], a performance toolkit for accurately measuring and pinpointing performance bottlenecks. HPCTOOLKIT uses novel techniques for measurement and analysis of parallel programs. In particular, it uses statistical sampling of hardware performance counters and attributes metrics to both the calling context in which they occur and program structure, including loops and inlined procedures. In this poster, we describe how we use HPCTOOLKIT to pinpoint scalability and multi-core performance bottlenecks. We demonstrate HPCTOOLKIT's utility by applying it to two SciDAC applications: the S3D code for simulation of turbulent combustion [2] and the MFDn code for *ab initio* calculations of microscopic structure of nuclei [3]. Although in this poster we focus on measurement and analysis to support manual tuning, the precise characterization of performance bottlenecks that we obtain is a prerequisite for more automatic approaches based on feedback-directed optimization.

2. Overview of HPCToolkit

HPCTOOLKIT [1] consists of components for measuring the performance of fully-optimized executables of parallel programs, analyzing application binaries to correlate measurements with program structure, and novel analysis techniques for pinpointing performance bottlenecks. We designed HPCTOOLKIT to:

- *Work at binary level for language independence.* This enables HPCToolkit to support measurement and analysis of multi-lingual codes with external binary-only libraries.
- *Profile rather than adding code instrumentation.* Sample-based profiling is less intrusive than code instrumentation and requires only very modest data volume.



HPCTOOLKIT workflow.

- *Collect and correlate multiple performance metrics.* Performance problems typically cannot be diagnosed with only one species of event.
- *Compute derived metrics to aid analysis.* Synthetic metrics, such as memory bandwidth consumed, often provide insight for optimization.
- *Attribute costs very precisely.* HPCTOOLKIT is unique in its ability to associate measurements with dynamic calling context, loops, and inlined code.

2.1. Pinpointing and quantifying performance bottlenecks

We have developed several novel performance analysis techniques that provide deep insights into application performance on parallel platforms.

2.1.1. Pinpointing scalability bottlenecks in SPMD codes. HPCTOOLKIT pinpoints scalability bottlenecks with an exquisite level of detail — all the way down to invidividual source lines in the program and the calling context in which the bottlenecks arose. Surprisingly, using our technique, one can do this for production runs of fully-optimized applications with run-time measurement overhead of only a few percent for arbitrary SPMD parallel programs including applications written using MPI [4]. Here, we briefly sketch how this process works.

Consider two executions of an application using weak scaling, 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 amount of 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 we would expect this to hold in *each part* of the application. Using HPCTOOLKIT, we measure the cost associated with each calling context in a program execution on each of p and q processors. We use a data structure called a *calling context tree* (CCT) to record our measurements. We expect that the execution time attributed to all pairs of *corresponding nodes*[1] in the CCT to be identical. We use this expectation as the basis for identifying where performance scalability falls short. A similar strategy can also be applied to analyze strong scalability [4].

[1] Corresponding nodes in a pair of CCTs represent the same calling context, *e.g.*, `main` calls `solve`, which calls `sparse matrix-vector multiply`.

2.1.2. Identifying performance bottlenecks in multithreaded codes. We are in the process of extending HPCTOOLKIT with two novel techniques for pinpointing, quantifying, and explaining performance bottlenecks in multithreaded programs. Our approach will support diagnosis of two types of inefficiencies: insufficient parallelism (which might be due to load imbalance or serialization) and parallelization overhead.

To pinpoint and quantify insufficient parallelism in executions of multithreaded programs, we will 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 parallel overhead, 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.

3. Case studies

Here we study the performance of two SciDAC codes to illustrate two uses of HPCTOOLKIT: analysis of multi-core scaling and analysis of overall performance on a single core.

3.1. Multi-core scaling of S3D

S3D uses direct numerical simulation (DNS) to study the micro-physics of turbulent reacting flows. The S3D code decomposes a 3D data volume among a collection of processes organized as a 3D logical mesh. Processes have same number of grid points and communicate using MPI. Most communication occurs between processes that are nearest neighbors in the 3D logical mesh.

To investigate the node performance of S3D, we studied two executions of an ethylene combustion simulation: one execution on a single core of a 1.9 GHz AMD quad-core Opteron 8347 processor (known as a Barcelona), and a second execution on all eight cores of a shared-memory node consisting of a pair of Barcelonas. Each of the cores has 512MB of private L2 cache, and 2MB of shared L3 victim cache. The nodes of the system serve as a proxy for the quad-core Barcelona nodes of the NCCS Cray XT4. For the model problem, on a single core, the code achieved .56 FLOPs/cycle; on the eight core configuration, the model problem achieved .20 FLOPs/cycle for each core, yielding a total performance of .80 FLOPs/cycle per quad-core processor. Thus, in the eight core configuration, each core achieved only 36% of the performance of a single core in isolation. However, four cores of a quad-core processor achieved a 45% higher FLOP rate than a single core in isolation.

To understand the difference between the single core and eight core executions, we used HPCTOOLKIT to collect a call path profile for each core for both the single-core and eight-core executions. Figure 1 compares the performance of the execution on a single core and the performance of one of the cores in the eight-core execution. This figure shows a screen capture from HPCTOOLKIT's `hpcviewer` user interface, showing a *flat view* that correlates performance metrics with each loop nest in S3D. The `hpcviewer` display is in three parts: a source code window, a pane full of performance metrics of various types, and a navigation pane that relates

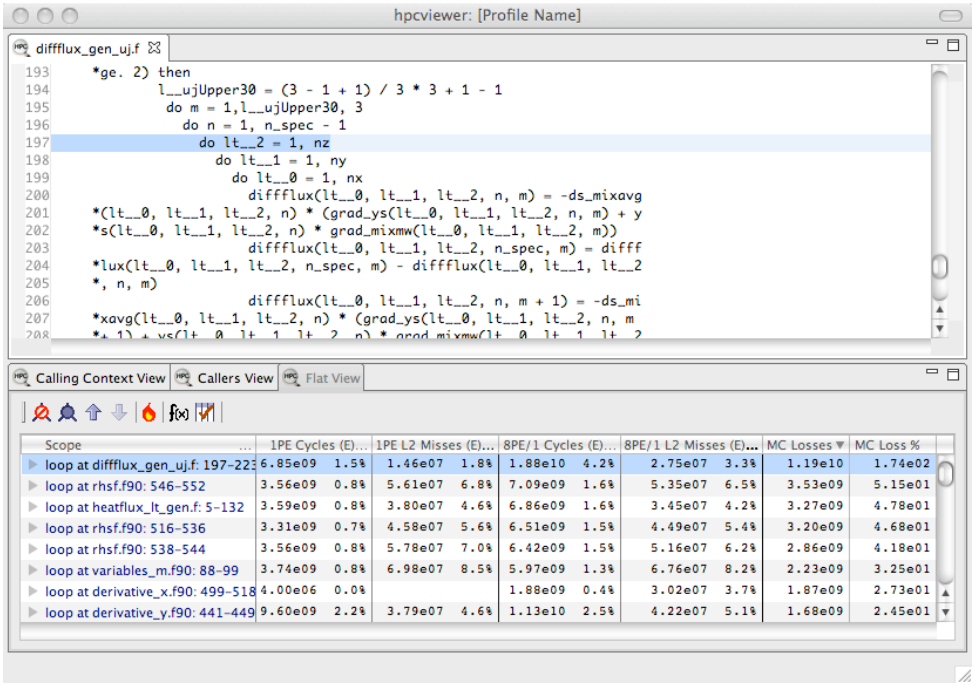


Figure 1. `hpcviewer` displaying a flat view of a timer-based call path profile for S3D.

values in the metric pane to the program source. The first three columns in the metric pane show cycles, L2 cache misses, and FLOPs for a single-core execution. The next three columns show the same metrics for one core of an eight-core execution. Each core is computing on a 30^3 volume of data. By comparing the cycles metric for one of the cores in eight-core execution with that of the single-core execution, we see that overall, the execution time on each core of the eight-core run is slower by a factor of 2.74 when all eight cores of the system are used. The last two columns of Figure 1 demonstrate `hpcviewer`'s capability for computing arbitrary derived metrics on demand. Column seven of the metric pane shows a derived metric representing the multicore loss of performance, computed as the difference between the cycles on one of the eight cores and the cycles of a single-core execution. The final column of the display shows the percentage loss of performance in each loop nest, written in scientific notation.

In Figure 1, the loops of the application are sorted according to their multicore loss, in descending order. The source pane shows part of the loop nest for which the loss was greatest. In the rightmost column of the metric pane, we see that its multicore loss for this loop was 174%. By comparing the L2 misses for one of the cores in an eight-core execution with those of a single core running in isolation, we see that the number of L2 misses increases by 88%. This loop, a diffusive flux calculation, is the most memory-intensive loop in the program. Inspection of the other top loops in the display shows multicore losses ranging from 24.5% to 51.5%. By sorting and ranking loop nests according to their multicore losses, `hpcviewer` focuses attention on the loop nests that deserve the most attention for performance tuning.

3.2. Single-core performance of MFDn

The "Many Fermion Dynamics — nuclear" code (MFDn) evaluates the many-body Hamiltonian and obtains the low-lying eigenvalues and eigenvectors using the Lanczos algorithm. The code shows good scaling and load balance on 15,000 cores, but is lacking in per-process efficiency.

To understand how much opportunity exists for improving the performance of the loop nests

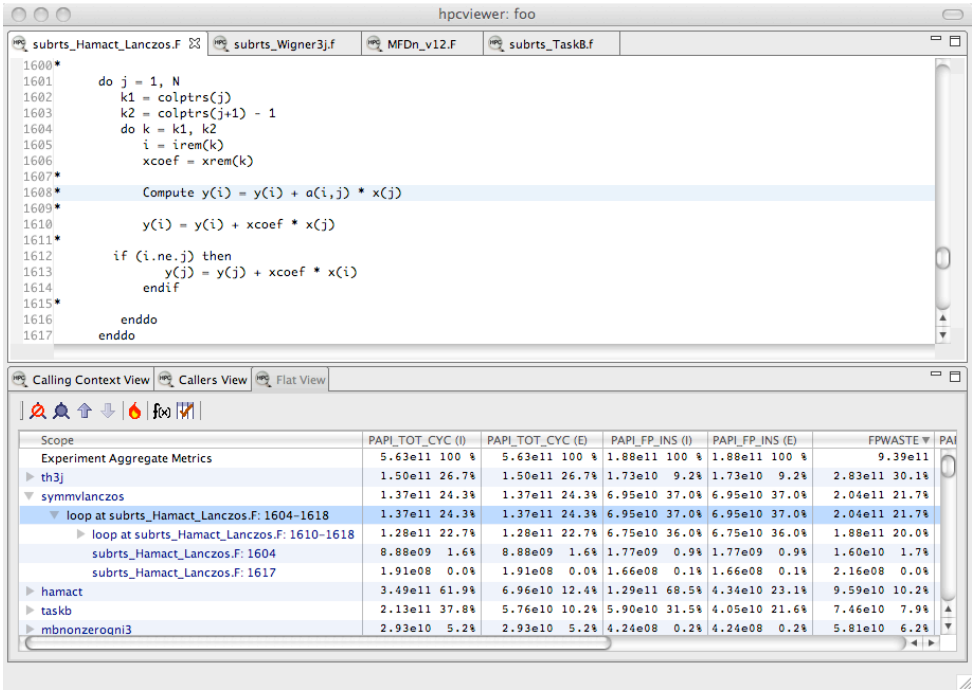


Figure 2. `hpcviewer` displaying floating point waste (FPWASTE) for the loop nest within MFDn's `symmvlanczos`.

in MFDn, we used HPCTOOLKIT to measure and analyze the code on one core of a 2.2 GHz dual-core Opteron. Figure 2 shows both measured metrics (cycles and FLOPs) and a derived metric FPWASTE. The FPWASTE column represents the difference between the theoretical peak FLOPs possible on the Opteron core and the actual FLOPs executed, *i.e.*, $(2 \times \mathrm{CYC}) - \mathrm{FP\_INS}$; this indicates how much unrealized opportunity for floating point computation is associated with each loop. The highlighted loop computes the quantity $\mathbf{y} = \mathbf{y} + \mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is represented in the *compressed sparse column* format. This loop accounts for roughly 20% of the FPWASTE in the execution. The matrix-vector multiply in the highlighted loop requires a load and store of a result value for each non-zero in the sparse matrix $\mathbf{A}$. If the matrix could be stored in *compressed sparse row format*, the calculation would perform one write and no loads of intermediate results per matrix row; this would improve the overall efficiency of this calculation.

4. Current status

Presently HPCTOOLKIT is being used to analyze application performance on Opteron-based Linux clusters. As soon as kernel bugs are resolved on the Cray XT and Blue Gene/P, these tools will become available on the DOE's leadership class machines.

References

[1] Rice University 2008 HPCToolkit performance tools. `http://www.hipersoft.rice.edu/hpctoolkit`
[2] Monroe D 2002 *SciDAC Review* URL `http://www.scidacreview.org/0602/html/combustion.html`
[3] Vary J P, Shirokov A M and Maris P 2008 *ArXiv e-prints* **804** (*Preprint* `0804.0836`)
[4] Coarfa C, Mellor-Crummey J, Froyd N and Dotsenko Y 2007 *ICS '07: Proceedings of the 21st Intl. Conference on Supercomputing* (New York, NY, USA: ACM) pp 13–22 ISBN 978-1-59593-768-1