Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles

Nathan R. Tallent, Laksono Adhianto, John M. Mellor-Crummey
Rice University
{tallent,laksono,johnmc}@cs.rice.edu

***Abstract*—Applications must scale well to make efficient use of today's class of petascale computers, which contain hundreds of thousands of processor cores. Inefficiencies that do not even appear in modest-scale executions can become major bottlenecks in large-scale executions. Because scaling problems are often difficult to diagnose, there is a critical need for scalable tools that guide scientists to the root causes of scaling problems.**

Load imbalance is one of the most common scaling problems. To provide actionable insight into load imbalance, we present post-mortem parallel analysis techniques for pinpointing and quantifying load imbalance in the context of call path profiles of parallel programs. We show how to identify load imbalance in its static and dynamic context by using only low-overhead asynchronous call path profiling to locate regions of code responsible for communication wait time in SPMD executions. We describe the implementation of these techniques within HPCTOOLKIT.

I. INTRODUCTION

Applications must scale well to make efficient use of today's class of petascale computers, which contain hundreds of thousands of processor cores. In fact, inefficiencies that do not even appear in modest-scale executions can become major bottlenecks in large-scale executions. Because scaling problems are often difficult to diagnose, there is a critical need for tools that guide scientists to the root causes of scaling problems.

Load imbalance is one of the most common scaling problems in Single Program Multiple Data (SPMD) scientific applications. Load imbalance in an SPMD execution with P processes is caused by an uneven distribution of work that forces some processes to idle between synchronization points.

To effectively diagnose load imbalance, performance tools for petascale platforms must accomplish three things. *First*, performance tools must provide actionable insight. By *actionable insight*, we mean insight into an application's performance that justifies concrete actions such as determining how to resolve a performance bottleneck or deciding that there are no significant and worthwhile opportunities for improving performance. In other words, the role of performance tools is not so much to highlight program hot spots, but to pinpoint and diagnose bottlenecks. *Second*, tools must measure accurately and with enough precision to pinpoint performance bottlenecks. Without accurate and detailed performance measurements of fully optimized applications, analysis is unproductive. *Third*, for performance tools to be useful on large-scale systems, measurement and analysis techniques must scale to hundreds of thousands of processor cores. For example, tools must be able to monitor large executions unobtrusively enough not to significantly change that execution's performance characteristics.

Several tools describe the *effects* of load imbalance. Two common techniques include (1) presenting a process-time diagram of inter-process communication and (2) showing the per-process distribution of costs for a particular routine that is assumed to be balanced. Relatively few tools analyze performance measurements to identify the root *causes* of load imbalance.

Nearly all tools that try to identify the causes of load imbalance analyze traces [14], [28]–[30]. The advantage of these methods is that temporal information can make it easy to distinguish between certain kinds of performance problems. For example, assuming synchronous communication, temporal information makes it easy to distinguish between local imbalance caused by a late receiver as opposed to a late sender. The key disadvantage of tracing is that it is difficult to scale because the size of each trace file is proportional to the length of an execution.

Another issue that affects both the measurement accuracy and the scaling potential of the above methods is that they collect trace data via instrumentation instead of asynchronous sampling. In contrast to asynchronous sampling, which has a controllable sampling rate [1], instrumentation is directly tied to code execution. In general, instrumentation-based measurement faces an inelastic tension between accuracy and precision. For instance, instrumentation of small frequently executing functions introduces overhead and generates as many trace records as function invocations. To make instrumentation-based measurements accurate, it is usually necessary to make coarse-grained and selective measurements [15]. Methods of reducing the amount of trace data collected or stored include 'lossy' online compression [4], [7]–[9], [12], lossless online compression [19], and feedback-directed selective tracing [31].

To avoid both the difficulties and drawbacks of scalable tracing, other tools collect *profiles* by collapsing an execution's temporal dimension. However, most profile-based tools do not attempt to perform any root-cause analysis of load imbalance. One exception to this is CrayPAT [6] which computes imbalance metrics that both provide a relative measure of and quantify load imbalance [5], [20]. CrayPAT's metrics suffer from two deficiencies. First, by relying on an assumption that may not be true, CrayPAT may overestimate the load imbalance of any procedure that does not begin and end with collective

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

synchronization. Second, CrayPAT's call graph profiles are based on static binary instrumentation. Accordingly, to obtain accurate measurements, it is usually necessary to make coarse-grained and selective measurements.

To provide actionable insight into resolving load imbalance, we make the following contributions:

- We show how to identify load imbalance in its static and dynamic context by locating procedures and loops responsible for communication wait time in SPMD executions. To ensure accurate, precise and scalable measurements, we use call path profiles collected by asynchronous call stack sampling that incur overhead of a few percent [25], [26]. Call *path* profiles are more precise than call *graph* profiles because they preserve unique calling contexts rather than merging them. Using post-mortem techniques, we attribute load imbalance at the loop level by augmenting call path profiles with static structure [25].
- We describe post-mortem parallel analysis techniques for interpreting and presenting call path profiles of parallel programs. In particular, we describe how to combine in parallel every thread-level call path profile in a large-scale execution into a summary call path profile.
- To help users understand the resulting summary, we present call path profiles in three complementary views: Calling Context (top-down), Callers (bottom-up), and Flat. We show how these views facilitate a top-down analysis methodology that enables one to quickly move from an execution-level summary to the thread-level detail for a region of imbalance. We also describe how the Callers and Flat views can be built from a Calling Context view — even when the Calling Context view only contains summary metrics defined with non-commutative and non-associative operators.
- We provide access to deferred thread-level data at the level of individual nodes in a call path profile and graph those thread-level values in a variety of formats.

We describe the implementation of these techniques within HPCTOOLKIT [1], [21], an integrated suite of tools for measurement and analysis of program performance on computers ranging from multicore desktop systems to supercomputers.

Because we base our work on call path profiles instead of tracing, our method cannot distinguish between as many root causes as a trace-based tool. However, at the same time, our measurement approach is more scalable than tracing. By combining loop-enriched call path profiles with our post-mortem analyses, we present the most precise root-cause load imbalance analysis that is available from profiles to date.

The rest of this paper is organized as follows. Section II describes our post-mortem technique for identifying load imbalance. Section III describes additional post-mortem analyses of call path profiles to enable our load imbalance analysis. Section IV explains how we present call path profiles to facilitate rapid analysis. In Section V, we use HPCTOOLKIT to analyze load imbalances. Finally, Section VI compares our approach with related work and Section VII summarizes our contributions and ongoing work.

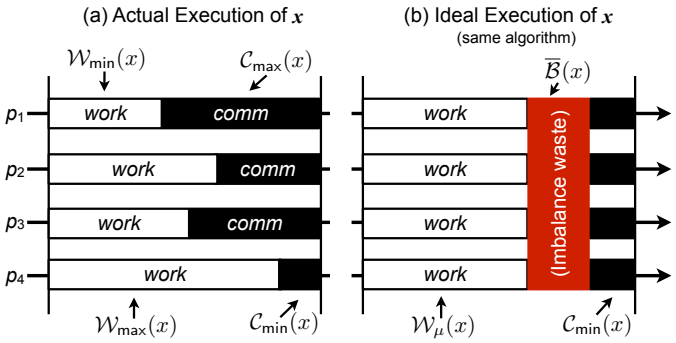

Fig. 1. Computing imbalance waste (red) in a synchronized execution of procedure x by comparing an (a) actual and (b) ideal execution (based on the same algorithm). Let $\mathcal{W}$ and $\mathcal{C}$ represent work and exposed communication, respectively; let μ represent arithmetic mean; and assume any interleaved work and communication has been aggregated.

II. IDENTIFYING LOAD IMBALANCE VIA POST-MORTEM BLAME SHIFTING

This section describes how to identify load imbalance in its static and dynamic context by locating procedures and loops responsible for communication wait time in SPMD executions.

A. Motivating Example

Assume an SPMD execution with P processes. Consider a procedure x in this execution that collectively synchronizes, performs some work and communication, and then collectively synchronizes again. Figure 1 shows a simplified breakdown of x's execution. The figure separates x's work ($\mathcal{W}$) from its exposed communication ($\mathcal{C}$) and consolidates the former on the left and the latter on the right of each time line. (We use the qualifier *exposed* to account for both blocking and non-blocking communication.) We would like to compute an imbalance waste ($\overline{\mathcal{B}}$) metric quantifying the magnitude of inefficiency due to load imbalance in procedure x's execution.

To compute this imbalance metric, we compare an actual and ideal execution of procedure x (based on the same algorithm). An ideal execution is both perfectly balanced and has no exposed communication waiting time. Assume we have summary metric values, based on wall clock time, that provide the arithmetic mean (μ), minimum (min) and maximum (max) and total (Σ) work and exposed communication across all P instances of x. In an ideal execution, each instance of x performs the same amount of work, or $\mathcal{W}_{\mu}(x)$ time units. Assume that the ideal communication time for each instance of x is the minimum value across each instance, which is $\mathcal{C}_{\mathsf{min}}(x)$ time units. Thus, in an ideal execution, each instance of x takes $\mathcal{W}_{\mu}(x) + \mathcal{C}_{\mathsf{min}}(x)$ units of time as opposed to $\mathcal{W}_{\mu}(x) + \mathcal{C}_{\mu}(x)$ units in the actual execution. The difference between the latter and former is imbalance waste:

$$\begin{aligned} \overline{\mathcal{B}}(x) &= (\mathcal{C}_{\mu}(x) - \mathcal{C}_{\mathsf{min}}(x))\, P \\ &= (\mathcal{W}_{\mathsf{max}}(x) - \mathcal{W}_{\mu}(x))\, P \end{aligned} \tag{1}$$

A nice property about this equation is that to compute the waste due to load imbalance, it is only necessary to consider

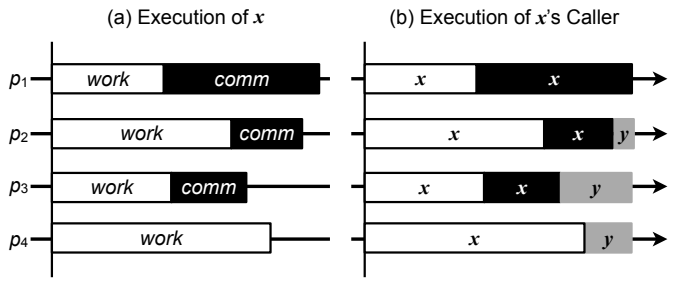

Fig. 2. (a) An execution of procedure x that is imbalanced may be (b) balanced at x's caller because of compensating 'backfill' by procedure y. (Assume any interleaved work and communication has been aggregated.)

either work or exposed communication. CrayPAT's imbalance metric is based precisely upon these observations [5], [20].

The most fundamental problem with applying Equation 1 to compute imbalance waste is that it assumes that a procedure performs collective synchronization on both entry and exit. Figure 2 shows a time-line view of the work and communication components of an execution of a different version of procedure x. Although the execution of x is imbalanced, the execution at the caller of x may actually be balanced because of compensating 'backfill' by another procedure y. In particular, we would expect backfill for point-to-point communication. Thus, although we could apply Equation 1 to the 'main' procedure, it would be misleading to use it to compute imbalance waste for procedure x.

To compute the waste due to load imbalance, we propose two refinements to the above technique. First, because the nice properties of Equation 1 depend on finding synchronization points in the computation, we show how to scalably find *balance points* in a call path profile. We then compute imbalance waste for every balance point. This refinement improves the accuracy of a tool by ensuring that it will not report misleading values for imbalance waste. Second, we develop a more accurate measure of the waste due to load imbalance. Equation 1 computes imbalance waste by assuming a procedure's minimum communication time is the ideal for every process in the execution. That is, it assumes (a) that every process had the same communication and computation requirements, which may not be true for point-to-point communication; and (b) that this minimum communication time itself had no exposed idling, which also may not be true. We show how to accurately compute imbalance waste by identifying *exposed idleness*. Then, to yield actionable insight, we attribute that idleness to balance points as an imbalance waste metric. This is an application of an approach that we call 'blame shifting' [27]. We discuss these refinements in turn.

B. Refinement: Find Balance Points In Call Path Profiles

To identify load imbalance in its full static and dynamic context, we use HPCTOOLKIT's call path profiler `hpcrun` to attribute execution costs of optimized executables to the full calling context in which they occur. `hpcrun` collects a call path profile using asynchronous sampling for each thread in an execution. To attribute metrics back to source code, HPCTOOLKIT's `hpcprof` tool combines a call path profile with program structure information reconstructed by a post-mortem analysis of an application's object code and its debugging sections [25]. Using this information, HPCTOOLKIT attributes metrics to dynamic call paths overlaid with static context such as loops and inlined functions. HPCTOOLKIT's measurement approach scales well to large executions because it is distributed (thread-based) and because profiles grow slowly over time. In particular, profiles grow only with the number of new calling contexts revealed on each sample.

HPCTOOLKIT's `hpcrun` profiler collects a separate call path profile for each thread in an application. It represents each call path profile using a data structure called a *calling context tree* (CCT) [2]. A CCT is a weighted tree whose root is the program entry point and whose leaves represent sample points. Given a sample point (leaf), the path from the tree's root to the leaf's parent represents a sample's calling context. Leaves of the tree are weighted with metric values such as sample counts that represent program performance in context.

To analyze the performance of a whole parallel execution, HPCTOOLKIT uses the post-mortem analyses described in Section III to create a canonical CCT that summarizes all of the individual thread-level CCTs within an execution. HPCTOOLKIT associates nodes in the canonical CCT with their original thread-level values and also computes derived aggregate metrics such as minimum, maximum, sum, mean, and standard deviation that summarize thread-level data. To distinguish costs based on static and dynamic nesting, HPCTOOLKIT computes *inclusive* and *exclusive* versions of each metric. Inclusive metrics for a particular node reflect costs for the entire subtree rooted at that node. Exclusive metrics for a procedure exclude costs for a procedure's callees.

To locate balance points in the canonical CCT, we define a CCT node x as balanced if every instance of x takes the same amount of time; or if for every pair of threads t and u in the execution, we have $\mathsf{time}(x,t) = \mathsf{time}(x,u)$. To develop a practical version of this test that is tolerant of measurement noise, we observe that, within the canonical CCT, the root node is expected to be the 'most balanced' node.[1] Therefore, by using a relative measure of balance, we can compare CCT nodes against the root node to gauge how balanced they are. One good relative measure of the balance of the cost at a node is its coefficient of variation (CV), i.e., its standard deviation divided by its mean. Using this metric, we say a node x is balanced if x is not a communication call and the following condition holds:

$$\mathsf{CV}_I(x) \leq \alpha \times \mathsf{CV}_I(\mathit{root}) \tag{2}$$

where *root* is the CCT root node and α is a small expansion factor such as 1.1 to account for the fact that the root node is expected to be the 'most balanced' node. The 'I' subscript indicates inclusive metric values.

[1]For example, `MPI_Finalize` is nearly always at the end of a program and usually performs a barrier.

C. Refinement: Shift Blame for Idleness

Recall that Equation 1 computes imbalance waste by assuming a procedure's minimum communication time is the ideal for each instance in the execution. We noted that this may not be true for point-to-point communication. It may also be the case that this minimum communication time itself has exposed idling, such as might occur in poor MPI implementations.

To accurately compute a metric that quantifies the waste of load imbalance, we first identify exposed idleness ($\mathcal{I}$) in communication calls.[2] One way to do this is to identify waiting routines by name. For example in Cray's implementation of MPI on its XT series platforms, the routine `MPIDI_CRAY_Progress_wait` is associated with exposed waiting. Another option is to tag instructions related to waiting and use post-mortem binary analysis to separate measurements into their distinct work, communication, and idling components [27]. In either case, the result is an accurate and precise measure of idleness during the course of an execution.

However, this is not enough. Locating exposed idleness identifies nodes in the canonical CCT that are *victims* of load imbalance. To obtain insight into the *causes* of load imbalance, it is better to shift the blame of this exposed idleness as close as possible to its cause (perpetrator) or possible causes (suspects). Unfortunately, without temporal information, it is not generally possible to identify the precise cause of idleness. The causes of idleness can be complex, ranging from imbalanced computation (that is possibly succeeded by a chain of idleness from communication dependencies) to a poorly scaling communication primitive. Moreover, an aggregate imbalance caused by several CCT nodes may not be readily apparent because the individual contributions of each node may not be additive.

Nevertheless, although we cannot directly identify the perpetrators of imbalance, we can accurately identify suspects by shifting the blame for idleness ($\mathcal{I}$) to its deepest balanced ancestor node. By definition, a balanced node cannot contribute to imbalance. Consequently, if a descendant of a balanced node contains idleness, that idleness must be caused by one or more descendants of that balanced node. This observation enables us to define a metric that accurately and insightfully represents the waste due to imbalance. We define both exclusive (E) and inclusive (I) versions of the metric. Given a CCT node x, a simplified version of the imbalance waste ($\overline{\mathcal{B}}$) that should be assigned to that node is as follows:

$$\overline{\mathcal{B}}_E(x) = \begin{cases} \sum_{c \in \mathcal{C}\text{-frontier}(x)} \mathcal{I}_{\Sigma,I}(c) & x \text{ is balanced} \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

$$\overline{\mathcal{B}}_I(x) = \overline{\mathcal{B}}_E(x) + \sum_{c \in \text{children}(x)} \overline{\mathcal{B}}_I(c) \quad (4)$$

[2]Our method assumes that that it is possible to identify exposed idleness using asynchronous sampling. This is easy to do if idleness takes the form of busy waiting. It is also possible to identify exposed idleness that takes the form of sleep waiting (blocking) if there is no overthreading.

In Equation 3, the term $\mathcal{I}_{\Sigma,I}(x)$ represents the sum (over all threads) of inclusive exposed idleness for node x.[3] The function $\mathcal{C}$-frontier(x) returns every communication-waiting node c that is a descendant of x and for which the path between x and c contains no other communication or balanced node. Equation 4 is the natural inductive definition for inclusive values.

To compute a relative measure of imbalance for a CCT node x, we simply divide by the aggregate inclusive imbalance of the root node. Although we have used total idleness ($\mathcal{I}_\Sigma$) to compute an imbalance metric, it is also possible to create variants based on mean, minimum and maximum idleness. To implement this blame shifting, we use a simple depth-first-search traversal over the canonical CCT.

III. Scalable Analysis of Call Path Profiles

To apply the post-mortem load imbalance analysis of Section II to a large-scale execution, it is necessary to have a canonical calling context tree (CCT) that summarizes all of the individual thread-level call path profiles within an execution. Creating a canonical CCT requires unioning each thread-level CCT in the execution so that a context appears in the canonical CCT if and only if it appears in some thread-level CCT. If an execution contains T threads and there are M metrics per thread, then *each node* in that execution's canonical CCT will have $T \times M$ associated metric values. Because for large-scale executions T can be on the order of hundreds of thousands (currently) to millions (near future), it is neither reasonable to process each CCT sequentially nor feasible to store all thread-level metrics in memory. To handle large-scale measurements, it is critical that analysis and presentation itself be scalable.

To support scalable analysis and presentation of call path profiles, we developed `hpcprof-mpi`, a parallel version of HPCTOOLKIT's `hpcprof` tool. Like `hpcprof`, `hpcprof-mpi` attributes measurements of executions to source code and creates a database that can be presented by HPCTOOLKIT's presentation tool, `hpcviewer`. Unlike `hpcprof`, it is parallel and scalable.

When given all per-thread measurements for a large-scale execution, `hpcprof-mpi` does two things. To scalably analyze and attribute measurements to source code, `hpcprof-mpi` creates a canonical CCT that summarizes a whole execution. This involves scalably creating metrics that summarize all per-thread CCT data. Then, to facilitate scalable presentation, it generates a database of thread-level data correlated with the canonical CCT. Our database's design enables HPCTOOLKIT's `hpcviewer` presentation tool to present an execution's summary data without concern for the scale of parallelism it represents.

A. An Overview of `hpcprof-mpi`

`hpcprof-mpi` is based on Single Program Multiple Data (SPMD) parallelism and is implemented with MPI [16]. The

[3]For simplicity, Equation 3 is based purely on tree structure. In practice, we compute exclusive metrics according to the hybrid definition of Section IV, where procedure frames are treated differently than static structure.

high-level algorithm can be divided into four key phases. Assume an `hpcprof-mpi` job executes with P processes.

First, the master process divides the thread-level call path profiles into P groups, and assigns one group to each process. Each process is responsible for processing all the thread-level profiles assigned to it.

Second, `hpcprof-mpi` creates a canonical CCT that represents the union of all thread-level call path profiles. The canonical CCT contains a context — a path from a leaf to the root — if and only if it appears in some thread-level CCT. Although the canonical CCT represents a union of all thread-level CCTs, in practice it does not grow linearly with the number of threads. The thread-level CCTs of SPMD scientific applications are often very similar. Even applications that model multiple physical systems usually do not induce more than a handful of distinct CCT groups, and the groups themselves have commonality between them. This means that with respect to its structure, we expect the canonical CCT to be no more than a small constant factor larger than the average thread-level CCT. To create the canonical CCT, `hpcprof-mpi` performs a parallel (tree-based) reduction on the thread-level CCTs. (Metric data is excluded from this reduction.) Then it uses a parallel (tree-based) broadcast to return the canonical CCT to each process. After this step is completed, each MPI process contains a copy of the canonical CCT. `hpcprof-mpi` aligns each thread-level CCT with the canonical CCT so that the canonical CCT can serve as an index for both the new summary metrics as well as all of the thread-level metrics.

Third, `hpcprof-mpi` computes a useful set of derived summary metrics such as minimum, maximum, sum, mean, and standard deviation to summarize the values of each thread-level metric. To compute a derived metric for a given canonical CCT node x, it is necessary to use the metric value from each thread's CCT node that corresponds to x. Since `hpcprof-mpi` cannot depend on storing all thread-level inputs to a derived metric computation in memory simultaneously, it computes derived metrics *incrementally*, a process that is discussed in the next subsection.

Finally, `hpcprof-mpi` generates a profile database that associates calling contexts in the canonical CCT with summary metrics. Each canonical call path can be used to index into a database of the detailed thread-local metrics. This database is conceptually a dense matrix, though in practice it is often sparse.

B. Scalably Computing Metrics

HPCTOOLKIT was originally designed to compute derived metrics with all thread-level data simultaneously resident in memory. For example, consider the case of computing the arithmetic mean for a particular node in an execution's canonical CCT. If the execution contained T threads, then the arithmetic mean $m(x)$ for a given CCT node x would be $\frac{1}{T}\sum_{t=1}^{T} m(x,t)$, where $m(x,t)$ represents node x's metric value for thread t. For executions on petascale systems, the number of threads can be very large and it is therefore not feasible to require all thread-level metric values for each CCT node to reside in memory simultaneously.

Algorithm 1: incrementally-compute-metrics: Incrementally compute derived metrics in parallel.

Input: Vector $\mathbf{X} = \langle x_1, x_2, \ldots, x_n \rangle$ of IN metric values.
Input: Metric descriptor $\mathbf{M}$, including (1) n_A accumulators (state variables), $a_1, \ldots, a_{n_A}$, of type $\mathbb{A}$; and (2) four sets of functions with the following names and prototypes (where i corresponds to accumulator i):

initialize	$\bigcirc_i()$	:	$\mapsto \mathbb{A}$
accumulate	$\bigodot_i(a, x)$	: $\mathbb{A} \times$ IN	$\mapsto \mathbb{A}$
combine	$\bigoplus_i(a_p, a_q)$	: $\mathbb{A} \times \mathbb{A}$	$\mapsto \mathbb{A}$
finalize	$\bullet(\langle a_1, \ldots, a_{n_A} \rangle, n)$	: $\mathbb{A}^{n_A} \times$ SIZE	$\mapsto$ OUT

Result: OUT value when metric descriptor $\mathbf{M}$ is applied to $\mathbf{X}$.

1 Divide $\mathbf{X}$ into P groups $\mathbf{X}_1, \ldots, \mathbf{X}_P$, one for each process.
2 *In parallel at each process* $p \in [1..P]$*:*
3 **let** $\mathbf{A}_p = \langle a_1, \ldots, a_{n_A} \rangle$ be accumulators for metric $\mathbf{M}$, where each $a_i = \bigcirc_i()$
4 **let** $\mathbf{X}_p = \langle x_1, \ldots, x_{n_p} \rangle$ be the metric values assigned to p
5 **foreach** x_i *in* $\mathbf{X}_p$ **do**
6 **foreach** a_j *in* $\mathbf{A}_p$ **do**
7 $a_j \Leftarrow \bigodot_j(a_j, x_i)$
8 *In parallel, reduce accumulators* $\mathbf{A}_2, \ldots, \mathbf{A}_P$ *into* $\mathbf{A}_1$. To reduce $\mathbf{A}_p$ and $\mathbf{A}_q$ into $\mathbf{A}_p$:
9 **foreach** $\langle a_{p,i}, a_{q,i} \rangle$ *in* make-pairs($\mathbf{A}_p, \mathbf{A}_q$) **do**
10 $a_{p,i} \Leftarrow \bigoplus_i(a_{p,i}, a_{q,i})$
11 **return** $\bullet(\langle a_1, \ldots, a_{n_A} \rangle, n)$

To address this problem, we have developed an approach to scalably compute derived metrics. We call the approach *incremental* because it tolerates receiving inputs one at a time rather than all at once. Linford et al. [14] observe that covariance can be computed incrementally. Our contribution is to formalize the technique and show how it can be applied to computing several kinds of metrics for scalably analyzing and presenting call path profiles. Specifically, we partition the thread-level inputs into chunks that can be as small as one. Any metric that can be expressed as a function of polynomials over thread-level data can be computed incrementally. This approach can also be used to compute the minimum and maximum, though it is insufficient to compute order statistics in general [23].

To compute a given metric m incrementally for a CCT node x, we use Algorithm 1. This algorithm takes as input a vector of values and a metric descriptor that represents a formula for computing an output metric value. The basic idea is to associate with each metric descriptor (1) a set of state variables called *accumulators* and (2) four sets of functions, whose prototypes are shown in the algorithm, for operating on those accumulators. Accumulators represent intermediate values obtained while applying a conventional metric formula that requires all input values simultaneously. The algorithm uses the functions associated with a set of accumulators to *initialize* the accumulators, to *accumulate* the input values, to *combine* the accumulators (for parallelism) and to *finalize* the accumulators to obtain the final output metric value.

Algorithm 1 divides the computation of an incrementally computed metric's value into stages, corresponding to these four function types. The key stages are accumulate and finalize. Recall that to compute the arithmetic mean for a particular CCT node x, we use the formula $\frac{1}{T}\sum_{t=1}^{T} m(x,t)$. To form the accumulate and finalize functions, we isolate portions of this formula that use commutative and associative operators from portions that do not; the latter operations are saved for the finalization stage. Thus, we associate the sum $\sum_{t=1}^{T} m(x,t)$ with an accumulator a_x and postpone the division by T to the finalization stage. This corresponds to an accumulate function of $a_x + m(x,t)$ and a finalization function of a_x/T.

We now consider the algorithm in more detail. Assume a parallel execution with P processes. First, line 1 partitions the input metric values into P groups, one group for each process, to facilitate data parallelism. Then, line 3 associates a set of accumulators with each group and initializes each accumulator using its initialize function. The accumulator in the arithmetic mean example is simply initialized to 0: $a_x = 0$.

Second, line 5 of the algorithm uses the metric's accumulate functions to form partially accumulated values for each process-local set of metric values. During this stage, each thread-level input is considered one by one and in no particular order. For each input value, the algorithm updates each accumulator using its respective accumulate function. For the arithmetic mean example, the one accumulator simply maintains a running sum of the input values. Consequently, when the arithmetic mean metric is given a new input $m(x,t)$, we update the accumulator using the following computation: $a_x \Leftarrow a_x + m(x,t)$.

Third, the algorithm uses a parallel reduction to reduce each set of local accumulators into one global set (line 8). This step relies on the metric's combine functions, which takes two accumulator values, a_{x_p} and a_{x_q}, and combines them into another accumulator value. The combine function for arithmetic mean is simply $a_{x_p} + a_{x_q}$.

Finally, line 11 of the algorithm applies the metric's finalize function to obtain its final value. The finalize function takes all of a metric's accumulators and the number of inputs and applies any additional operations that are required to obtain the metric's final value. The finalize step for arithmetic mean computes the final value $m(x)$ for node x using the operation a_x/T, where T is the number of thread-level inputs.

Algorithm 1 is scalable because instead of simultaneously requiring storage for all input values, it only requires simultaneous storage of one input value and a set of accumulators. Although as written this algorithm operates over one vector $\mathbf{X}$ of inputs, in practice, we apply it to a set of vectors, namely, one input vector per CCT node. For this case, $\mathbf{X}$ becomes an input matrix where row i is an input vector for CCT node i.

Figure 3 defines metric descriptors that are compatible with Algorithm 1 for mean, minimum, and standard deviation. Other metrics such as sum and maximum are similar. Each descriptor gives the number of accumulators needed and the definitions for the corresponding initialize $\bigcirc$, accumulate $\bigodot$, combine $\bigoplus$ and finalize $\bullet$ functions. Note that care must be taken when computing the minimum value for a metric over a set of processes. If no sample is taken at a particular location in a particular process, it is usually natural to represent this fact with 0; but this would lead to the minimum value over all processes being 0, an artificial result. Instead, it is usually more informative to compute the minimum *observation* by distinguishing observed and non-observed values.

Mean		Minimum	
$\bigcirc()$	$= 0$	$\bigcirc()$	$= \infty$
$\bigodot(a,x)$	$= a + x$	$\bigodot(a,x)$	$= \min(a,x)$
$\bigoplus(a_p,a_q)$	$= a_p + a_q$	$\bigoplus(a_p,a_q)$	$= \min(a_p,a_q)$
$\bullet(\langle a\rangle,n)$	$= a/n$	$\bullet(\langle a\rangle,n)$	$= a$

Standard Deviation		
$\bigcirc_i()$	$= 0$	
$\bigodot_1(a_1,x)$	$= a_1 + x^2$	$a_1 : \sum x^2$
$\bigodot_2(a_2,x)$	$= a_2 + x$	$a_2 : \sum x$
$\bigoplus_i(a_p,a_q)$	$= a_p + a_q$	
$\bullet(\langle a_1,a_2\rangle,n)$	$= \sqrt{(a_1/n) - (a_2/n)^2}$	

Fig. 3. Metric descriptors, compatible with Algorithm 1, for (a) mean and minimum using one accumulator a; and (b) standard deviation using two accumulators a_1 and a_2.

IV. Scalable Presentation of Call Path Profiles

To enable insightful presentation of performance data, we wish to present contextual measurements in multiple views: the Calling Context, Callers, and Flat views. A Calling Context view attributes performance metrics to their full calling context. If the Calling Context view looks down a call chain, the Callers view looks up a call chain to apportion metrics of a callee on behalf of its calling contexts. A Flat view organizes performance data according to an application's static structure so that all costs incurred in any calling context by a procedure are aggregated together. Each view is important in analysis. However, the Calling Context view is foundational in the sense that it can be used to define both the Callers and Flat views, but not vice versa.

In this section, we describe formal definitions for metric values in `hpcviewer`'s Calling Context, Callers and Flat views. We designed these definitions for two purposes. First, with appropriate definitions, it is possible to compute the Callers and Flat views not just from thread-level CCT metrics but also from *derived* CCT metrics. The significance of this is that to create all three views with derived metrics, `hpcprof-mpi` only needs to generate one view — a Calling Context view with derived metrics. Second, our definitions correct deficiencies of informal definitions used in prior versions of HPCTOOLKIT. For example, our definitions correctly account for recursion when computing metrics for the Callers and Flat view.

A. Calling Context view

The Calling Context view is represented by the canonical calling context tree (CCT) that `hpcprof-mpi` generates. An important feature of the canonical CCT is that it is a fusion of dynamic calling contexts and static program structure. Each node in the CCT can be classified as representing either a

dynamic or static scope. A *dynamic* node is either a call site or a statement, where a statement is a sample point. A *static* node is either a procedure frame, loop, or alien code, where the latter usually represents inlined procedures.

We define two types of Calling Context metrics: inclusive and exclusive. Inclusive metrics for a particular node reflect costs for the entire subtree rooted at that node. Exclusive metrics are of two flavors, depending on the node type. For a procedure frame, exclusive metrics reflect all costs within that procedure but excluding callees. In other words, for a procedure frame, costs are exclusive with respect to dynamic call chains. For all other scopes, exclusive metrics reflect costs for the scope itself; i.e., costs are exclusive with respect to static structure. This hybrid definition makes sense when we consider that although we often think of procedure frames in the context of call chains, it is natural to think of loops in the context of a procedure.

To scalably compute summary metrics for this view, we use a straightforward adaptation of Algorithm 1 [23].

B. Flat view

`hpcviewer`'s Flat view organizes performance data according to an application's static structure. This means that all costs incurred by a procedure in any calling context are aggregated together. This view 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. Because we create this view from calling-context sensitive metrics, it has exclusive *and* inclusive variants of each metric.

As with the Calling Context view, it is trivial to compute derived metrics for the Flat view by first creating all thread-level (Flat view) metrics. However, creating all thread-level metrics for petascale executions is both time- and space-consuming. We have developed an algorithm to compute a Flat view with derived metrics using a Calling Context view with (non-finalized) derived metrics [23].[4]

C. Callers view

If the Calling Context view emphasizes the caller-callee relationship, the Callers view emphasizes the callee-caller relationship. If the Calling Context view looks down a call chain, the Callers view looks up a call chain to apportion metrics of a callee (in its context) on behalf of its caller. Consequently, the Callers 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. The Callers view quickly highlights which calling contexts to `MPI_Wait` are important.

When several levels of the Callers view are expanded, saying that the Callers view apportions metrics of a callee on behalf of its caller can be ambiguous: what is the caller and what is the callee? To resolve this ambiguity we can say that the Callers view apportions the metrics of a particular procedure *in its various calling contexts* on behalf of that context's caller. Alternatively but equivalently, the Callers view apportions the metrics of a particular procedure on behalf of its various *calling contexts*.

As with the Calling Context and Flat views, it is trivial to compute derived metrics for the Callers view by first creating all thread-level (Callers view) metrics. However, if this was undesirable for the Flat view, it is even more undesirable for the Callers view. Creating all Callers-view thread-level metrics is especially time- and space-consuming because the Callers view is quadratic in terms of the Calling Context view. To see this, observe that the Callers view is a forest, with a root for each procedure frame. If the Calling Context view has n nodes, then *each* root x in the Callers view potentially has $O(n)$ nodes, because (procedure frame) x could appear in every calling context within the CCT. To avoid such behavior, we have developed an algorithm to compute individual nodes of a Callers view with derived metrics *on demand* using a Calling Context view with only (non-finalized) derived metrics [23].

V. Case Study

To demonstrate the ability of our techniques to identify load imbalance in executions of emerging petascale applications, we apply them to study the performance of PFLOTRAN. PFLOTRAN is a code for modeling multi-phase, multi-component subsurface flow and reactive transport using massively parallel computers [13], [18]. The code is designed to predict the migration of contaminants underground. "PFLOTRAN solves a coupled system of mass and energy conservation equations for multiple compounds and phases including H_2O, supercritical CO_2, black oil, and a gaseous phase" [18]. With support from the DOE SciDAC program, the authors of PFLOTRAN plan to use it to understand radionuclide migration at the DOE Hanford facility and model sequestration of CO_2 in deep geologic formations. Typical simulations involve massive computation due to ten or more chemical degrees of freedom on a grid of millions of nodes. PFLOTRAN employs the PETSc [3] library's Newton-Krylov solver framework.

We ran PFLOTRAN on 8184 cores of the Cray XT5 partition of Jaguar, located at Oak Ridge National Laboratory's National Center for the Computational Sciences. Each XT5 node contains two hex-core 2.6 GHz Opterons (Istanbul). The PFLOTRAN test problem was a steady-state groundwater flow problem in heterogeneous porous media on an $850 \times 1000 \times 80$ element discretization with 15 chemical species per cell and 2 billion degrees of freedom. We used HPCTOOLKIT's `hpcrun` to simultaneously collect two hardware counter metrics: cycles and floating point operations, where the latter is a measure of work. `hpcrun`'s effective sampling rate was 230 samples/second; its overall overhead was 3% (relative to PFLOTRAN's unmonitored execution time of 15.3 minutes); and it generated 7.2 GB of data (0.9 MB/process). `hpcprof-mpi`

[4] A non-finalized derived metric has not had its finalize function applied.

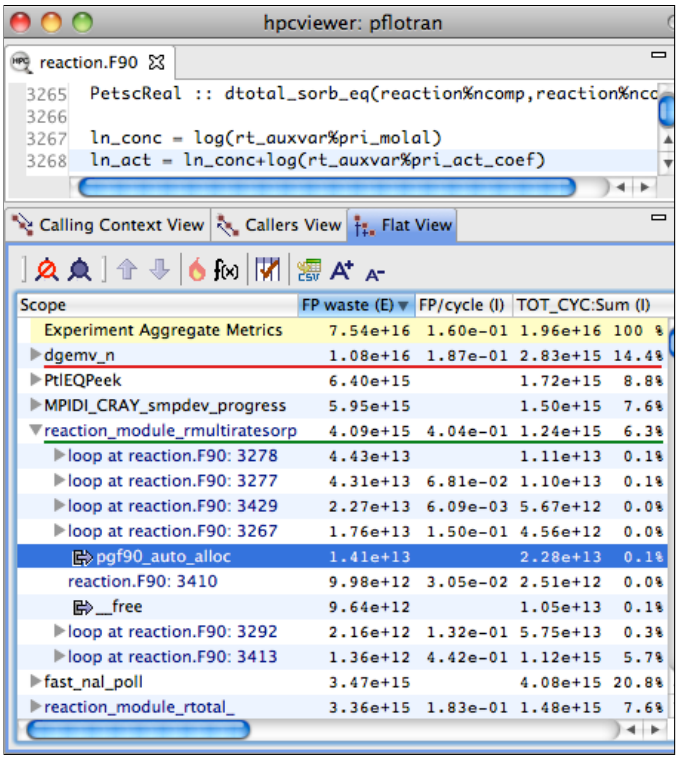

Fig. 4. A Flat view of PFLOTRAN's floating point waste.

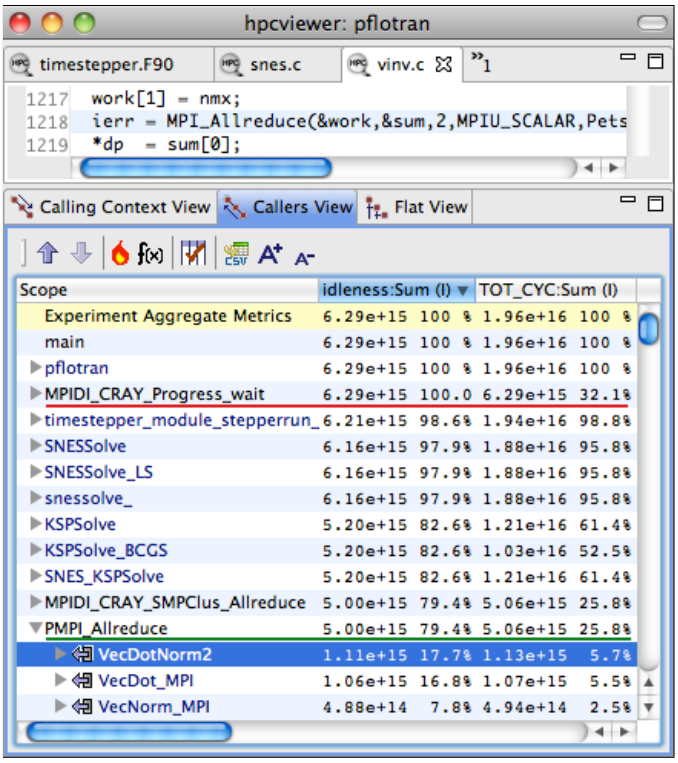

Fig. 5. A Callers view of PFLOTRAN's idleness.

processed the measurements in 8.0 minutes using 48 cores and generated a 3.6 GB database. Using 24 or 96 cores, `hpcprof-mpi` generated the same database in 13.0 or 5.5 minutes, respectively.

We first assess the overall floating point efficiency of PFLOTRAN's execution. Figure 4 shows a Flat view of PFLOTRAN's static structure with two metrics that highlight floating point utilization. The first metric, which is the sort key, provides a measure of floating point waste ('FP waste'). Each Opteron core on an XT5 node has a maximum peak performance of four double-precision floating point operations (FLOPs) per cycle. Therefore, we can compute floating point waste by subtracting actual floating point throughput from ideal throughput as follows: $(4 \times \text{cycles}) - \text{FLOPs}$. The presentation tool computes this 'FP waste' metric using the cycle and FLOPs summary metrics that `hpcprof-mpi` generates by summing over all processes in the execution. This metric is exclusive, meaning that it excludes callees (hence the 'E' modifier). The second metric is inclusive FLOPs per cycle. Overall, this execution of PFLOTRAN performed 0.160 floating point operations per cycle, which is only 4.0% of peak.

The first routine that the 'FP waste' metric highlights is `dgemv_n`, which is underlined with red in Figure 4's lower pane. Although this matrix-vector multiply routine consumes 14.4% of the execution's cycles, it has a floating point efficiency of 0.187 FLOPs/cycle. For comparison, the matrix-matrix multiply routine `dgemm_kernel` (not shown) delivers 2.28 floating point operations per cycle. This low efficiency bears further investigation.

Figure 4 also shows static structure within the `reaction_module_rmultiratesorption` routine (underlined with green), which accounts for 6.3% of the total cycles. At 0.404 FLOPs/cycle, this routine has better floating point throughput than `dgemv_n`. The static structure recovered by HPCTOOLKIT exposes two important compiler transformations. The first is shown in the highlighted call site to `pgf90_auto_alloc`. This call site indicates that the Portland Group (PGI) compiler automatically allocated a temporary vector to implement the highlighted Fortran 90 statement shown in the source code (top) pane. The right hand side of this statement performs a vector logarithm and then adds the result to another vector. Although the whole statement could have been implemented with a loop and without a temporary, HPCTOOLKIT shows that the compiler allocated an unnecessary temporary vector. The second transformation of interest is that the four of the five top loops in this routine are actually compiler-generated scalarization loops to implement Fortran 90 vector operations.

Figure 5 shows a Callers view of PFLOTRAN sorted by total inclusive idleness, summed over all processes ($\mathcal{I}_{\Sigma,I}$ from Equation 3). The right metric column labeled 'TOT_CYC:Sum (I)' shows the corresponding inclusive sum of cycles across all processes. It is immediately apparent that 32.1% of the execution ($6.29 \times 10^{15} / 1.96 \times 10^{16}$) is consumed by idleness.

To determine where this idleness is manifested, we simply scan down the 'idleness:Sum (I)' column. We can immediately see that waiting on the Cray XT occurs within the low-level `MPIDI_CRAY_Progress_wait` routine (underlined with

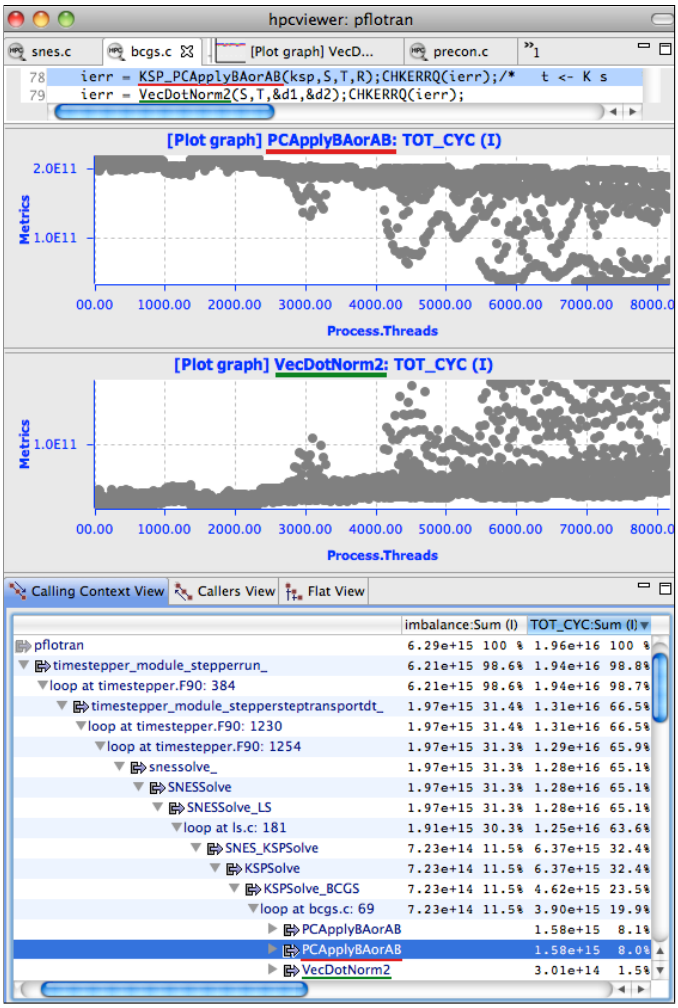

Fig. 6. A Calling Context view of PFLOTRAN's load imbalance.

red) that is part of the XT's MPI library. Skipping the next few high-level routines representing the time-stepper loop and PETSc's non-linear solvers (variants of `SNESSolve`), we arrive at `MPI_Allreduce` (underlined with green). The 'idleness:Sum (I)' metric column indicates that 79.4% of the total idleness of this execution occurs within `MPI_Allreduce` or one of its callers.

The Callers view shows all the callers of `MPI_Allreduce` during the execution (that registered samples); the Figure shows the top three — `VecDotNorm2`, `VecDot_MPI`, and `VecNorm_MPI` — which account for 42.3% (17.7 + 16.8 + 7.8) of the execution's idleness.

We now focus on obtaining a clearer picture of source of load imbalance. Figure 6 shows a Calling Context view of PFLOTRAN with both an inclusive imbalance metric ('imbalance:Sum (I)') that corresponds to $\overline{\mathcal{B}_I}$ in Equation 4 and an inclusive cycles metric, 'TOT_CYC:Sum (I)'. The bottom pane shows the hot path expanded according to the imbalance metric and through five loops, the last of which is at `bcgs.c:69`.

This latter loop, which calls several routines, is within PETSc's BCGS KSP solver. The second and third most time-consuming routines that the loop calls are `PCApplyBAorAB` and `VecDotNorm2` (the latter of which was highlighted in Figure 5). The top (source code) pane shows that the former is called before the latter. The two middle panes show scatter plots of the thread-level values for each routine, respectively. Within the scatter plot of `PCApplyBAorAB`, one can observe a clear pattern of minima that are complemented by a nearly identical pattern of maxima in `VecDotNorm2`'s scatter plot. The differences of these minima and maxima from their peers are very similar. By descending further down the call path or by looking up the call path of the Callers view, one can confirm that both of these routines call `MPI_Allreduce`. In other words, `PCApplyBAorAB`'s scatter plot shows a pattern of processes that finish early, only to wait within `VecDotNorm2`.

To determine at least in part the source of this imbalance, we can descend down the hot call path of `PCApplyBAorAB` (with respect to 'TOT_CYC:Sum (I)') to a parallel matrix multiply routine, which uses a sequential version of the same, `MatMult_SeqBAIJ_N`. The FLOPs scatter plot for this sequential routine shows the same pattern of minima as in `PCApplyBAorAB`'s cycle scatter plot, confirming that a few processes have very little work to do.

While this case study of an execution of PFLOTRAN is not an exhaustive analysis, that was not our intent. However, the case study does illustrate how having aggregate load imbalance and idleness metrics attributed to each node of an execution's canonical CCT can help pinpoint, quantify, and understand sources of performance loss.

VI. Related Work

We know of no other profiling-based tool, besides CrayPAT [6], that attempts to compute a measure of the severity of load imbalance for large-scale parallel applications. CrayPAT's relative and absolute imbalance metrics [5], [20] are essentially reflected by Equation 1. As indicated in Section II, among other things, Equation 1 can be misleading if a procedure does not perform collective synchronization on both entry and exit. To address the accuracy of these imbalance metrics and to increase their interpretive power, we shifted the blame of explicit idleness to balance points in a canonical call path profile. Additionally, because we base our work on HPCToolkit's asynchronous-sampling-based call *path* profiling, our metrics can be more accurate and precise than CrayPAT's binary-instrumentation-based call *graph* profile. In particular, we can attribute metrics to static structure in its full calling context for a run-time overhead of only a few percent.

Paradyn is unique among root-cause-analysis tools by using dynamic instrumentation to perform an online performance bottleneck search [17], [22]. However, because its Performance Consultant has a centralized implementation that queries application processes for data, its online analysis is not fundamentally scalable.

Nearly all other tools to identify the root causes of load imbalance use instrumentation-based tracing. The basic dis-

advantages of these approaches are that tracing is difficult to scale and instrumentation-based measurement faces an inelastic tension between accuracy and precision.

The most recent work on scalable load balance analysis has been within the context of SCALASCA [10]. SCALASCA scalably analyze traces to identify temporal patterns that reflect inefficiency, such as late sending or receiving [10], [30]. Although scalable and post-mortem, this analysis is based on a trace replay engine and thus can potentially be time-consuming for long executions of many processes. To help reduce the overhead of tracing, SCALASCA provides the option of selectively tracing based on information from a prior profile [31]. Since this feedback-directed trace reduction requires an additional execution, it is likely impractical in some situations.

Linford et al. recently proposed a technique based on SCALASCA's measurement and analysis abilities [14]. After collecting a trace, they sample it (post-mortem) to establish correlations between procedure times and wait times. They then form (causal) hypotheses from these correlations and test their hypotheses via replay simulation of trace file. It is not clear how practical this proposal is or how effective it is compared to other methods.

There are several existing techniques for root-cause analysis, but they face scalability challenges with respect to both measurement and analysis. Vetter automatically classifies MPI communication inefficiencies such as late receivers and senders using decision tree classification, a machine learning approach [28]. Because the decision tree is generated using microbenchmarks, the approach is extensible and portable. Meira et al. difference execution paths between two synchronization points to identify the possible causes of waiting time [29]. Both of these methods, as described, serially process full trace files post-mortem.

Because traces are so useful but so difficult to scale, other work focuses on improving the ability of instrumentation-based tracing tools to visualize the victims of imbalance. ScalaTrace relies on program analysis to losslessly compresses traces by compactly representing certain commonly occurring trace patterns [19]. Gamblin et al. have explored techniques for dynamically reducing the volume of trace information [7]–[9]. They report impressively low overheads, but they also, in part, use selective instrumentation that results in coarse measurements. Lee et al. reduce trace file size by using scalable k-Means clustering to select representative data [11], [12]. However, because their data-reduction technique is post-mortem rather than online, it assumes all trace data is stored in memory. This is feasible only with coarse-grained and selective tracing. Others have explored (manual) selective tracing based on application characteristics [4]. In summary, although all of this work enables better trace collection, it does not directly address how to enable root-cause analysis or obtain performance insight.

VII. Conclusions

We have demonstrated that it is possible to obtain insight into load imbalance by using call path profiling and scalable post-mortem analyses of the resulting profiles. To assist root-cause analysis of load imbalance in parallel executions, we have shown how to shift the blame of the idleness caused by load imbalance up call paths to its potential causes. We have described several techniques for scalably analyzing and presenting call path profiles, including (a) computing metrics incrementally to summarize a large-scale execution; (b) presenting context-sensitive metrics in three complementary views; and (c) linking context-sensitive summary metrics for a whole execution to their individual thread-level values so that they can be graphed in different forms.

We have based our work on asynchronous-sampling-based call path profiling. Consequently, we were able to both measure execution performance for very little overhead and yet collect very precise calling-context-sensitive metrics. With this foundation, we were able to present load imbalance and other inefficiency metrics in their static and dynamic contexts.

We have exclusively focused on profiling because, in contrast with traces, profiles do not grow with time but only with the number of unique contexts that a sample reveals. However, in large-scale parallel applications, some scalability problems are related to patterns of waiting that are not readily distinguishable with only a profile. To distinguish between different types of temporal bottlenecks, it is necessary to incorporate time into HPCToolkit's measurements. One approach we are investigating is collecting asynchronous-sampling-based call path traces [1]. To collect such a trace, one simply maintains both a calling context tree (CCT) and a series of small (12 byte) records with each sample represented by a CCT node id and a time stamp. We expect this to enable HPCToolkit to collect extremely rich trace information at large scales for much less overhead than instrumentation-based approaches.

In prior work, we have developed online approaches for blame shifting in shared-memory applications [24], [27]. The advantage of online techniques is that they can directly blame waiting on its causes without recording full temporal information. However, these techniques are only useful if they do not cause a tool to itself to become a bottleneck. We are exploring possibilities for online blame shifting in the context of large-scale distributed systems.

Acknowledgments

Development of HPCToolkit is supported by the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25762. This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. HPCToolkit would not exist without the contributions of Michael Fagan and Mark Krentel.

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," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[2] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proc. of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 1997, pp. 85–96.

[3] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.0.0, 2008.

[4] I.-H. Chung, R. E. Walkup, H.-F. Wen, and H. Yu, "MPI performance analysis tools on Blue Gene/L," in *Proc. of the 2006 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 123.

[5] L. DeRose, B. Homer, and D. Johnson, "Detecting application load imbalance on high end massively parallel systems," *Lecture Notes in Computer Science*, vol. 4641/2007, pp. 150–159, 2007.

[6] L. DeRose, B. Homer, D. Johnson, S. Kaufmann, and H. Poxon, "Cray performance analysis tools," in *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2008, pp. 191–199.

[7] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed, "Scalable load-balance measurement for SPMD codes," in *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[8] ——, "Clustering performance data efficiently at massive scales," in *Proc. of the 24th ACM International Conference on Supercomputing*. New York, NY, USA: ACM, 2010, pp. 243–252.

[9] T. Gamblin, R. Fowler, and D. A. Reed, "Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes," in *Proc. of the 2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–12.

[10] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[11] C. W. Lee and L. V. Kalé, "Scalable Techniques for Performance Analysis," Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, Tech. Rep. 07-06, May 2007.

[12] C. W. Lee, C. Mendes, and L. V. Kalé, "Towards Scalable Performance Analysis and Visualization through Data Reduction," in *Proc. of the 13th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Miami, Florida, USA, April 2008.

[13] P. Lichtner *et al*, "PFLOTRAN project web site," https://software.lanl.gov/pflotran, 2009.

[14] J. C. Linford, M.-A. Hermanns, M. Geimer, D. Boehme, and F. Wolf, "Detecting load imbalance in massively parallel applications," Forschungszentrum Jülich, Tech. Rep. FZJ-JSC-IB-2008-09, December 2008.

[15] A. D. Malony, S. Shende, A. Morris, and F. Wolf, "Compensation of measurement overhead in parallel performance profiling," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 2, pp. 174–194, 2007.

[16] *MPI: A Message Passing Interface Standard*, Message Passing Interface Forum, June 1999, http://www.mpi-forum.org/docs/mpi-11.ps.

[17] 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," *Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[18] R. T. Mills, C. Lu, P. C. Lichtner, and G. E. Hammond, "Simulating subsurface flow and transport on ultrascale computers using PFLOTRAN," *Journal of Physics Conference Series*, vol. 78, no. 012051, 2007.

[19] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalatrace: Scalable compression and replay of communication traces for high-performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 8, pp. 696–710, 2009.

[20] H. Poxon and L. DeRose, "Addressing scalability with the Cray performance analysis toolset," Workshop on Performance Tools for Petascale Computing, Center for Scalable Application Development Software, July 2009. [Online]. Available: http://cscads.rice.edu/workshops/summer09/slides/performance-tools/CScADS09-PoxonDeRose.pdf

[21] Rice University, "HPCToolkit performance tools," http://hpctoolkit.org.

[22] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proc. of the 2003 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 21.

[23] N. R. Tallent, "Performance analysis for parallel programs: From multicore to petascale," Ph.D. dissertation, Department of Computer Science, Rice University, March 2010.

[24] 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*. New York, NY, USA: ACM, 2009, pp. 229–240.

[25] 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*. New York, NY, USA: ACM, 2009, pp. 441–452.

[26] 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*. New York, NY, USA: ACM, 2009, pp. 1–11.

[27] N. R. Tallent, J. M. 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*. New York, NY, USA: ACM, 2010, pp. 269–280.

[28] J. Vetter, "Performance analysis of distributed applications using automatic classification of communication inefficiencies," in *Proc. of the 14th International Conference on Supercomputing*. New York, NY, USA: ACM, 2000, pp. 245–254.

[29] J. Wagner Meira, T. J. LeBlanc, and A. Poulos, "Waiting time analysis and performance visualization in carnival," in *Proc. of the 1996 SIGMETRICS Symposium on Parallel and Distributed Tools*. New York, NY, USA: ACM, 1996, pp. 1–10.

[30] F. Wolf, B. Mohr, J. Dongarra, and S. Moore, "Automatic analysis of inefficiency patterns in parallel applications," *Concurrency and Computation: Practice and Experience*, vol. 19, pp. 1481–1496, Feb. 2007, special Issue *Automatic Performance Analysis*.

[31] B. J. N. Wylie, M. Geimer, and F. Wolf, "Performance measurement and analysis of large-scale parallel applications on leadership computing systems," *Sci. Program.*, vol. 16, no. 2-3, pp. 167–181, 2008.