Flexible Data Aggregation for Performance Profiling

David Böhme, David Beckingsale and Martin Schulz
Lawrence Livermore National Laboratory,
Livermore, California 94551

***Abstract*—Almost all performance analysis tools in the HPC space perform some form of aggregation to compute summary information of a series of performance measurements, from summations to more complex operations like histograms. Aggregation not only reduces data volumes and consequently storage space requirements and overheads, but is also crucial to extract insights from recorded measurement data. In current tools, however, most aspects that control the aggregation, such as the data dimensions to be reduced, are hard-coded in the tool for a set of particular use cases identified by the tool developer and cannot be extended or modified by the user. This limits their flexibility and often results in users having to learn and use multiple tools with different aggregation options for their performance analysis needs.**

We present a novel approach for performance data aggregation based on a flexible key:value data model with user-defined attributes, where users can define custom aggregation schemes in a simple description language. This not only gives users the control to deploy the particular data aggregation they need, but also opens the door for aggregations along application-specific data dimensions that cannot be achieved with traditional profiling tools. We show how our approach can be applied for performance profiling at runtime, cross-process data aggregation, and interactive data analysis and demonstrate its functionality with several case studies driven by real world codes.

I. Motivation

Understanding the performance of modern HPC applications in increasingly complex execution environments is a difficult task. To learn more about an application's performance, developers insert measurement probes or use monitoring techniques to observe and measure the application's behavior and then analyze the acquired data for performance bottlenecks. However, recording an application's behavior in full detail can quickly generate unmanageable amounts of data, while the desired performance *insights* can often be summarized in just a few numbers or statistics, such as the total time spent in certain program locations. Therefore, data aggregation – computing summary information from a series of measurement records – is an important mechanism for performance analysis tools. Depending on the chosen analysis task, different reduction functions need to be deployed, which can mean computing simple scalar values (e.g., sums or average values) to detailed but compact representations of the input value distribution (e.g., histograms).

In the context of performance analysis, aggregation is typically used on-line during the data recording stage—to reduce the amount of data that needs to be stored—in combination with further aggregation off-line for post-processing to uncover insights from the recorded pre-aggregated data. One typical scenario for performance profilers is to aggregate measurement data at runtime to report the total amount of wall-clock runtime spent in a function and process, which is then—at first—presented by a visualization tool as a further aggregation across all processes using averages. Unlike database query engines, the first set of on-line aggregations do not operate on large, existing data stores, but rather use *streaming reduction operations* to aggregate new measurement records immediately as they are produced. While this is necessary to not having to store the large amounts of data modern tools could produce, it also means that the initial aggregation decides what data is later available for the in-depth analysis.

It is therefore essential to find the matching aggregation algorithm for a particular performance analysis task that represent the right tradeoff between data volume and expressiveness. This tradeoff, however, strongly depends on the application's intrinsic performance characteristics as well as the analysis task at hand. For example, is there substantial performance variation over time that needs to be retained during the on-line aggregation? Is it sufficient to learn the total time spent in a given function, or do invocations from different program locations need to be distinguished, requiring any aggregation to retain spatial information?

In the realm of databases and general-purpose data analytics solutions, users can describe complex aggregation schemes in languages like SQL. This amount of flexibility has, so far, been unavailable for performance profiling: current state-of-the art HPC profilers only implement a single or few pre-defined aggregation schemes. Specifically, the criteria (or *dimensions*) by which performance information can be distinguished are hard-coded, and often limited to some form of program location (e.g., call path) and process/thread id. Users cannot add new criteria or modify the existing ones, let alone add application specific criteria. This shortcoming is often a result of the tools' limited data model or file formats, which is static and cannot be extended.

In this paper, we present a novel, unified approach for on-line and off-line performance data aggregation. Based on a flexible key:value data model with user-defined attributes, we introduce an abstract aggregation model that lets users define custom aggregation schemes by choosing aggregation keys, variables, and operations in a common aggregation description language. These aggregation schemes can then be applied to on-line performance profiling as well as off-line querying, providing users with a new level of flexibility. Specifically, we make the following contributions:

- An overview of different aggregation use cases
- A description language for configuring performance-data aggregation schemes
- An implementation of the aggregation system in Caliper, a flexible performance monitoring framework, for on-line

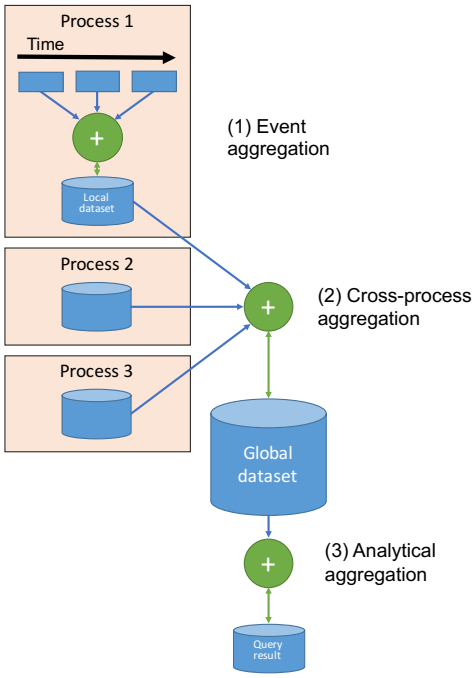

Figure 1. Data aggregation in different stages of an HPC performance analysis workflow.

performance profiling, cross-process data aggregation, and post-mortem data analysis
- A comprehensive overhead study of the aggregation operations implemented in Caliper

This customizable system allows us to create performance profiles and query results on any level of detail, covering the entire space between full traces and a scalar value representing a performance result using a single, unified data aggregation language. We demonstrate this functionality on a series of uses cases covering traditional aggregation operations as well as application specific aggregation. These case studies highlight that our approach brings a new level of expressiveness to runtime performance profiling that has traditionally not been available in HPC tools.

II. Data Aggregation Workflows

In the context of this paper, "data aggregation" refers to the process of computing a compact (typically constant-size) result from an arbitrary number of input values using reduction operators. Reduction operators can produce single values (e.g., sum, minimum, or maximum) or more complex structures such as histograms (i.e., a set of bins counting the input values within a certain value range).

Data aggregation occurs in different stages of a typical HPC performance analysis workflow, which we can distinguish by method (on-line or off-line) and application. Figure 1 shows three common applications: *event aggregation* to aggregate measurement events on a single process (or thread) over time, *cross-process aggregation* to aggregate data across all ranks of a parallel program, and *analytical aggregation* to extract insights from previously collected performance data. Either of these applications can run on-line, processing data directly from the target program as it is running, or off-line, processing previously recorded performance data from storage.

A. Event aggregation

Typically, event aggregation is used to calculate summary information over time, such as the total number of times a code region of interest was visited, or the total amount of runtime spent there. Event aggregation can be applied off-line, e.g., to summarize data from a previously recorded event trace. However, a much more common use case is on-line event aggregation, where a profiler accumulates performance measurements from a target program while the program is running. On-line aggregation generally uses *streaming reduction* to minimize memory and storage requirements: input values are never stored but immediately aggregated into the result, discarding the original input data.

In a profiling run, performance measurements can be triggered by instrumentation (inserting measurement probes directly into the target program, either in the source code or by modifying the binary program), or by system-generated interrupts in the form of time-based or event-based sampling (e.g., after a certain number of memory accesses or CPU instructions were executed).

B. Cross-process aggregation

For parallel programs, it is desirable to collect performance data for each process (and/or thread) individually. Indeed, for many performance analysis tasks, e.g., studying communication overhead or load imbalance, we need to compare and aggregate performance data from all of the target program's processes.

As the amount of generated performance data grows linearly with the number of processes, cross-process aggregation for large-scale performance experiments generally requires solutions that are scalable themselves. Scalable off-line solutions thererore require parallel programs in their own right, while on-line solutions may use dedicated data reduction network such as MRNet [1] or CBTF [2]. These networks utilize additional nodes in a cluster system to perform cross-process data recuction in parallel to a running target application.

C. Analytical aggregation

Data analysis is typically the final stage of a performance analysis experiment, where analysts extract insights from previously collected performance data. Here, aggregation is used to summarize data, calculate statistics, and present insights in a compact form. Commonly, analytical aggregation tools focus on data visualization and interactive exploration. Some specialized HPC-specific performance analysis tools provide graphical report explorers, such as TAU's PerfExplorer [3] or Score-P's Cube [4], for analytical aggregation. Data analysis functionality is also available through many general-purpose databases and visualization/data analytics toolkits.

On-line analytical aggregation (in-situ analysis) of performance data is somewhat rare, but used in some cases, e.g., for dynamic load balancing and auto-tuning.

D. Technical requirements

While the underlying aggregation concepts are similar, the technical requirements imposed on practical implementations for the aggregation use cases may differ substantially. On-line aggregation solutions generally run alongside the target application, often as a library within the same address space, and therefore share the computational and memory resources of the target program. Consequently, they must be implemented with miminmal computational and memory overhead. Operations that perturb the target application's original performance characteristics, such as I/O or extraneous synchronization between processes or threads, should be avoided. Minimal computational overhead is especially important for on-line event aggregation, which may be invoked frequently during the execution of the target program. For on-line cross-process aggregation, per-process performance data is typically buffered, with the cross-process reduction taking place at the end of the target program's execution, during uncritical sections, or in a separate process or thread.

Often, the choice between on-line or off-line aggregation for a particular task is a tradeoff between storage space and computational requirements: on-line solutions avoid intermediate I/O but must minimize computational overhead, whereas off-line solutions can perform expensive computations at the cost of additional I/O.

As our discussion above shows, data aggregation is a crucial capability that is used for a wide variety of tasks in different stages of a performance analysis workflow. With this wide range of use cases, many aggregation parameters depend on the specific analysis task at hand. In existing performance tools, however, aggregation schemes are instead determined by static design constraints, which limits the potential use cases the tool can cover. To alleviate this problem, we need a framework that gives control over aggregation schemes back to the user.

III. A General Aggregation Model

As a prerequisite for a general data aggregation framework, we have developed a common model to create customized aggregation schemes for all data aggregation applications in the performance analysis workflow. A key ingredient for this capability is a flexible, user-extensible data model. In the following, we outline the basic concepts of this data model and introduce our aggregation model.

A. A Key:Value Model for Performance Data

Other than most traditional performance profilers which rely on custom data models for their aggregation technique, our approach is based on a fully flexible key:value data model. A record in our model consists of a set of *attributes*, each representing a user-defined key:value pair. Attributes can have string, integer, or floating-point values. Each attribute has a unique *label* that serves as an identifier. The meaning of each attribute is defined by the user.

As an example, a typical record may look like this:

```
{ 'callpath'       : 'main/foo'
  'loop'           : 'mainloop'
  'loop.iteration' : 17
  'time.duration'  : 251 }
```

This record contains four attributes with a mix of string and integer values. Here, 'callpath', 'loop', 'loop.iteration', etc. are the user-defined attribute labels with their corresponding values captured at a specific point during program execution. System-provided attributes with, e.g., performance measurement values such as the 'time.duration' attribute can be added to a record as well. The contents of records are highly flexible: subsequent records in a given sequence can have entirely different sets of attributes.

B. Customizable Aggregation Schemes

Based on the general key:value data model, our aggregation approach allows users to create custom aggregation schemes. An aggregation scheme defines

- *Aggregation attributes*, which defines *which* attributes should be aggregated,
- an *aggregation key*, which defines *over* which attributes should be aggregated, and
- *aggregation operators*, which define the reduction functions that should be used.

These aggregation schemes can be used to configure any aggregation application (event aggregation, cross-process aggregation, and analytic aggregation), and target both on-line or off-line processing methods.

To formulate aggregation schemes, we devised a description language that borrows its basic syntax from SQL. Our language uses the following clauses:

- `AGGREGATE`: Specifies aggregation operators and attributes in a function-style syntax as comma-separated lists of `op(label)`, where `op` is a reduction operator and `label` identifies the aggregation attribute(s) that should be aggregated with the given operator.
- `GROUP BY`: Specifies the aggregation key as a comma-separated list of attribute labels. This clause works similar to its counterpart in SQL.

To illustrate the use of aggregation schemes in practice, consider the example program in Listing 1. The program calls functions `foo` and `bar` in a loop, with multiple function invocations in each loop iteration. Both the loop and the functions have been annotated with `mark_begin`/`mark_end`, which capture function names in the "function" attribute and the loop iteration in the "loop.iteration" attributes, respectively. Furthermore, we assume that our annotation system gathers performance metrics, such as time duration, for each begin/end interval. With this setup we can devise a number of different aggregation schemes to answer different performance questions. Consider the following example:

```
AGGREGATE count, sum(time)
GROUP BY  function, loop.iteration
```

This aggregation scheme that counts the number of records with a similar key (using the `count` operator) and sums up `time.duration` values using the `sum` operator. Results will be grouped by values of the `function` and

Listing 1. An example program with source-code annotations, exporting the current function names in the "function" attribute, and the main loop iteration number in the "loop.iteration" attribute.

```
void foo(int i) {
  mark_begin("function", "foo");
  // ...
  mark_end("function", "foo");
}

void bar(int i) {
  mark_begin("function", "bar");
  // ...
  mark_end("function", "bar");
}

int main() {
  for (int i = 0; i < 4; ++i) {
    mark_begin("loop.iteration", i);
    foo(1);
    foo(2);
    bar(1);
    mark_end("loop.iteration", i);
  }
}
```

`loop.iteration` attributes. These attributes form the aggregation key. The result of this scheme is a time-series function profile, which computes the total time spent in each function separately for each iteration of the annotated loop. We can print the result in a table:

```
function loop.iteration count sum#time
                                1       10
                          0     3       40
foo                       0     2       20
bar                       0     1       10
                          1     3       40
foo                       1     2       20
bar                       1     1       10
(...)
```

Here, the table columns represent the key and aggregation attribute labels. There is one table entry (row) for each unique aggregation key, i.e., each unique combination of `function` and `loop.iteration` values that were encountered in the annotated program (here, we only show the entries for `loop.iteration=0` and `1`). Note that the result includes separate entries for events where only one or none of the key attributes were set.

To obtain a more compact result that does not distinguish entries for each loop iteration, we can remove the `loop.iteration` attribute from the aggregation key:

```
AGGREGATE count, sum(time)
GROUP BY  function

function  count  sum#time
             13       130
foo           8        80
bar           4        40
```

In this way, custom aggregation schemes allow us to easily create different tradeoffs between data volume and detail.

IV. Design and Implementation

We have implemented our aggregation model in two separate, complementary components of our HPC performance data collection framework Caliper. The first component resides in Caliper's runtime library and handles on-line event aggregation. The second component resides in Caliper's off-line data analysis / query application, and performs off-line cross-process aggregation and analytical aggregation. Both use the same aggregation description language. In the following, we outline the design of these components in detail.

A. Background: Caliper

Caliper [5] is a general-purpose abstraction layer for performance introspection. Caliper provides a variety of application instrumentation and data-collection mechanisms, including a source-code instrumentation API, interrupt-based sampling, call-stack unwinding, and hardware performance counter access. These mechanisms work as independent building blocks, which can be combined at runtime through a callback API. Users specify which building blocks to use in a runtime configuration profile, either in a configuration file or environment variables.

Underneath, Caliper uses the flexible key:value data model outlined in Section III-A. When a Caliper-enabled target application runs, the instrumentation API and other selected data collectors can independently update attributes on a globally visible data structure, called *blackboard buffer* within Caliper. At any time, a *snapshot* can be triggered; this can happen synchronously with blackboard updates, explicitly through the Caliper API, or asynchronously, e.g., via timer signals. Caliper then creates a snapshot record with a compressed copy of the current blackboard contents at the particular time, and information about the triggering event. Performance measurements, such as timestamps or time durations, can also be added. Caliper then passes the snapshot record to a processing module, where it can be written to a trace or aggregated.

Caliper does not target any specific parallel runtime and can be used with any HPC programming model. The runtime system is thread-safe: all snapshots are processed in the thread that triggered them. In distributed-memory applications (e.g., MPI programs), Caliper creates per-process data sets. It does not perform any inter-process communication at runtime; cross-process data aggregation is performed in a postprocessing step.

B. On-line event aggregation

We have implemented on-line event aggregation as a service module for the Caliper runtime library. Figure 2 shows the basic on-line aggregation workflow. On startup, the aggregator service reads the aggregation key and aggregation attributes from a user-provided runtime configuration profile, and registers a callback function to process Caliper snapshot records. Snapshots can be triggered at runtime through a variety of means, including instrumentation hooks (e.g., on function entry or exit) or sampling timers. The exact snapshot

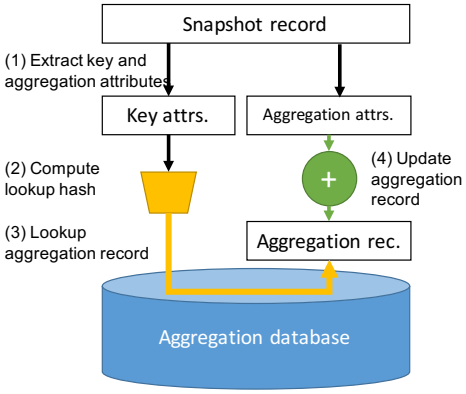

Figure 2. Snapshot processing workflow within the Caliper aggregation service.

triggering mechanism is independent of the aggregator service. The resulting snapshot records are inputs for the aggregation service.

Conceptually, a Caliper snapshot record is a set of independent key:value attributes as shown in Section III-A. When the aggregator service receives a snapshot record, it first extracts the key and aggregation attributes. The aggregator service maintains an in-memory database where it keeps an aggregation record for each unique set of key attributes. After extracting the key attributes, it computes a compact, collision-free hash value. This hash value is used to look up the corresponding aggregation record in the database, which contains the intermediate reduction results of the corresponding aggregation attributes. Once located, the aggregation record is updated by applying the reduction operators on the selected aggregation attributes. So far, our implementation provides sum, minimum, maximum and count operators. After the target program terminates, we flush the aggregation database by iterating over all entries, reconstructing the key attributes from the lookup hash value, and appending the reduction results from the aggregation record.

We maintain a separate aggregation database for each monitored thread of the target application, as this design avoids the use of thread locks. However, this design prevents the aggregation of values across threads at runtime (that is, we can compute aggregation results individually for each thread, but not a total result across all threads). Similarly, the aggregator service does not perform inter-process communication, and therefore cannot aggregate values across processes at runtime. Currently, aggregation across either threads or processes requires a post-processing step. Our implementation is async-signal safe, so it can be used in time-based or event-based sampling experiments.

C. Scalable cross-process aggregation

In addition to the on-line aggregation service module, we have also implemented our aggregation model in a query application that can merge and aggregate Caliper datasets, such as the per-process (or per-thread) datasets produced by a parallel program. Moreover, we have built a scalable MPI-based version of this query program that can run a query across many datasets in parallel. We use the query program both for cross-process and analytical aggregation.

The query program takes a query expression specifying the aggregation key and attributes, and a set of Caliper data files as input. In the MPI version, each process is assigned a subset of the data files, and first applies the query on its assigned dataset. Then, we organize the processes in a tree based on their rank, and perform a logarithmic reduction: "leaf" processes send the local aggregation results to their parent, where the partial results are aggregated again. The scheme continues on the next level of the tree until we reach the root process.

For the local aggregation on each process, we use the same approach as the on-line implementation as shown in Figure 2. However, instead of being generated by snapshots, the input records are now taken from the input files assigned to the process.

V. Evaluation

In this section, we discuss the performance and overheads of our aggregation approach. We first evaluate the runtime overhead of our on-line aggregation implementation in Subsection V-B, and then discuss the performance and scalability of our parallel post-mortem query application in Subsection V-C.

A. Test environment

Our experiments ran on Quartz, a 2 634-node cluster system at LLNL with Intel OmniPath interconnect, dual 18-core Intel Xeon E5-2695 2.1GHz processors, and 128 gigabytes of memory per node. All test programs and the Caliper library are built with the Intel 16.0.3 compilers using `-O2` optimization, and use MVAPICH2 MPI version 2.2.

B. On-line aggregation overhead

To evaluate the on-line aggregation overhead, we have conducted experiments with the CleverLeaf structured-grid shock hydrodynamics mini-application. We added Caliper source-code instrumentation in CleverLeaf to capture computational kernels, the AMR refinement level, main loop iterations, and some user-defined source-code regions. We also enabled MPI interception through the MPI profiling interface to capture MPI functions and the MPI rank. In total, we collected 7 attributes.

Because the aggregation scheme can impact the overhead, we examine three aggregation schemes with different aggregation keys. In the first (scheme A), the aggregation key contains all attributes except of the iteration number; in the second (scheme B), the aggregation key contains only two attributes; and the third (scheme C) contains all attributes including the main loop iteration number. Moreover, we examine two snapshot-collection modes: an asynchronous sampling mode where Caliper collects a snapshot every 10 milliseconds, and an event mode where Caliper collects a snapshot at each annotation event (i.e., begin/end of annotated regions). We compare the aggregation configurations against tracing (where we simply store every snapshot record) and against a baseline with no performance data collection. To obtain the evaluation results, we ran a test problem with the instrumented CleverLeaf program for 100 timesteps on 36 MPI ranks, occupying

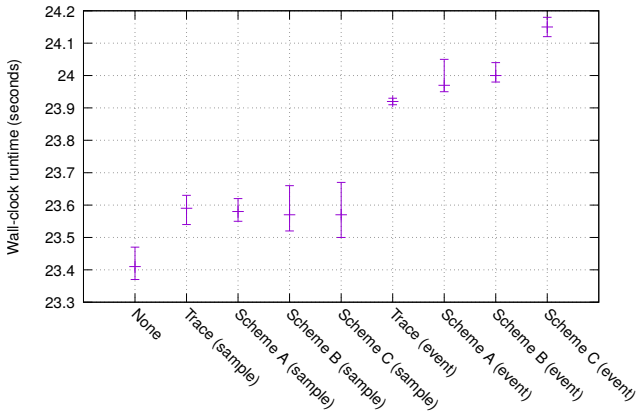

Figure 3. On-line aggregation overheads with different aggregation schemes in sampled and event-based performance data collection modes, compared to tracing and a baseline configuration without data collection. The bars show the average wall-clock runtime and runtime variation over 5 runs for each configuration.

all cores of a single cluster node, and measured the wall-clock runtime. We ran each configuration 5 times to quantify the run-to-run variation.

Figure 3 shows the overhead results, and Table I shows the number of snapshots and output records for each configuration. In the asynchronous sampling modes, our test program generates about 2360 Caliper snapshots per process. The runtime overhead over the baseline execution time is small (0.85%), and there is no measurable runtime difference between tracing and the different aggregation schemes.

In the synchronous event-based modes, our test program generates 219 382 snapshots per process, or about 9 200 per second. Accordingly, with 2% to 3.3%, the observed runtime overheads over the baseline execution are now slightly higher than in the sampling modes. The tracing configuration shows a small but measurably lower runtime overhead than the aggregation configurations. This is expected: tracing collects more data, but is computationally simpler. Aggregation schemes A and B result in 266 and 26 output records on each process, respectively. There is no measurable runtime difference between these two schemes. In contrast, scheme C, which creates separate performance data records for each main loop iteration, produces significantly more output records and has a noticably higher runtime overhead. However, the resulting profile data

Config	Snapshots	Output records
Trace (sample)	2 360	2 360
Scheme A (sample		67
Scheme B (sample		26
Scheme C (sample)		877
Trace (event)	219 382	219 382
Scheme A (event)		266
Scheme B (event)		26
Scheme C (event)		6 749

Table I
NUMBER OF SNAPSHOTS AND OUTPUT RECORDS (AGGREGATION RESULTS) PER PROCESS FOR TRACING AND DIFFERENT AGGREGATION SCHEMES.

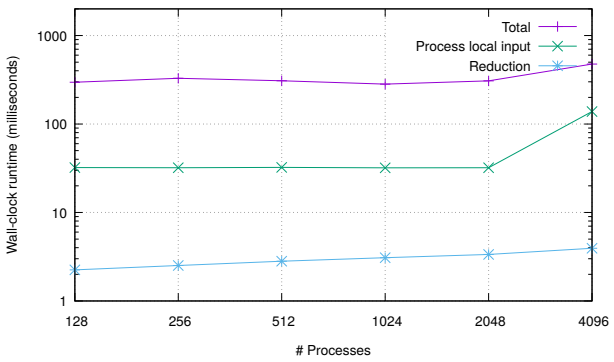

Figure 4. Scalability of cross-process aggregation in Caliper's MPI-based query application: total runtime (including I/O), reading and processing process-local input, and cross-process reduction.

in our case is still 32 times smaller than the full trace, while preserving dynamic performance information.

C. Scalability

We evaluate the scalability of our MPI-based off-line query application on a distributed Caliper dataset collected from ParaDiS, a dislocation dynamics application [6], using 4096 MPI processes on the Quartz cluster. The dataset contains a per-process time-series profile over computational kernels, MPI functions, MPI rank, and main loop iterations, with visit count and aggregate runtime of each unique region. Each of the 4096 input files contains 2174 snapshot records. For our performance evaluation, we let the parallel query application compute the total CPU time spent in computational kernels and MPI functions across MPI ranks, producing 85 output records. We run the query application with different process counts on subsets of the input dataset in a weak-scaling mode (i.e., constant problem size per process), always processing one input file per query process, and measure the wall-clock runtime on rank 0.

Figure 4 shows the results. The top line shows the total runtime (including I/O), the middle line shows the time for reading and processing process-local input, and the bottom line shows the time for the tree-based cross-process reduction. The time for reading and processing local input is constant over different process counts (as expected for a weak-scaling experiment), but can be affected by file I/O delays, as in the 4096-process case. As expected, the cross-process reduction time scales logarithmically with the number of processes. Overall, our tree-based reduction approach fulfills its scalability expectations.

VI. CASE STUDY: CLEVERLEAF

To show how flexible performance data aggregation can be used in practice, we present a case study with the CleverLeaf mini application.

A. Experiment setup

CleverLeaf is a structured-grid shock hydrodynamics mini-application with Adaptive Mesh Refinement (AMR) using

the SAMRAI toolkit [7] from Lawrence Livermore National Laboratory. The primary goal of CleverLeaf is to evaluate the application of AMR to Lagrangian-Eulerian hydrodynamics, and as a proxy to investigate the performance of AMR for structured Lagrangian-Eulerian hydrodynamics at scale. AMR is a computational technique used to increase the accuracy of a simulation in the areas of a problem where it is most effective, adding additional levels of mesh with increasing resolution over areas of interest. Applying more computational resources to select parts of the domain means both application runtime and memory usage can be reduced.

For our study, we examine the simulation of a version of the triple point shock interaction problem presented by Galera et al. in [8]. In this problem, a shock generates a large amount of vorticity and creates a complex area of interest which will be covered with a large number of subdomain patches by the AMR algorithm. We run the simulation with a coarse resolution of 640 by 240, and three levels of refinement, using 18 MPI processes on one node of LLNL's Quartz cluster.

We have added Caliper source-code annotations in CleverLeaf to obtain a detailed view of the program state, specifically:

- Names of computational kernels,
- user-defined source-code regions of interest, e.g. to delineate intialization, I/O, and computational phases,
- the AMR mesh refinement level,
- the simulation timestep (main loop iteration number).

We also obtain MPI function annotations and the MPI rank from Caliper's MPI wrapper. Performance data aggregation is applied in two stages: first, we use on-line event aggregation during the simulation run to collect per-process profiles, then we dive into specific aspects of the collected profiles interactively using off-line cross-process and analytical aggregation with the post-mortem query application.

B. Performance overview: computational kernels

Our first experiment is a common performance analysis scenario: we want to perform a low-overhead measurement run that gives us an estimate of the amount of time we spend in each of CleverLeaf's computational kernels. To do so, we use a sampling mode with a 100Hz sampling frequency, and set up the on-line aggregation to count the number of samples per computational kernel with the following simple aggregation scheme:

```
AGGREGATE count GROUP BY kernel
```

We then aggregate the per-process results with the off-line query application to obtain the total number of samples over all processes:

```
AGGREGATE sum(aggregate.count)
GROUP BY kernel
```

Note that we now use a `sum` reduction to accumulate the sample counts, which are provided by the on-line aggregation service in the `aggregate.count` attribute. Given the 100Hz sampling frequency, we can compute the (approximate) amount of CPU time spent in each kernel from the number of accumulated samples. The results are shown in Figure 5. We find that most of the annotated computational kernels except for `calc-dt` do not take much time. We also find that most samples were accumulated outside of the annotated kernels, including other computation as well as communication and I/O.

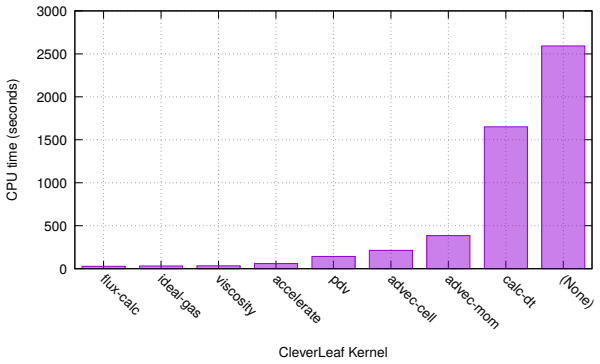

Figure 5. Profile of user-annotated computational kernels in CleverLeaf.

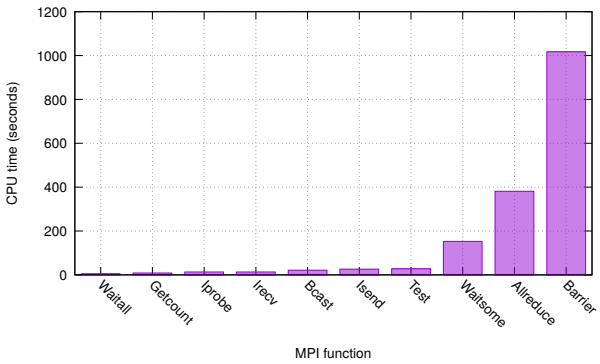

Figure 6. MPI function profile of CleverLeaf

C. Communication overhead

Our second experiment is another common performance analysis scenario: investigating communication overhead. For this example, we intercept MPI calls using Caliper's MPI wrapper, and aggregate visit counts and time spent in each MPI function per process with the following aggregation scheme:

```
AGGREGATE count,time.duration
GROUP BY  mpi.function
```

Figure 6 shows the accumulative CPU time for the top 10 MPI functions in CleverLeaf. By far most of the MPI time is spent in barrier synchronizations, followed by allreduce reductions. The time spent in point-to-point communication is comparatively small.

D. Load balance

To study load (im)balance, we need to compare values across processes (and threads). We accomplish this by including the MPI rank in the aggregation key:

```
AGGREGATE time.duration
```

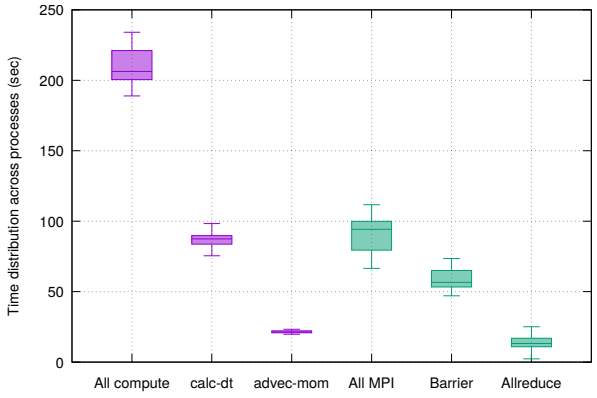

Figure 7. Time distribution for computational kernels and MPI functions across MPI ranks.

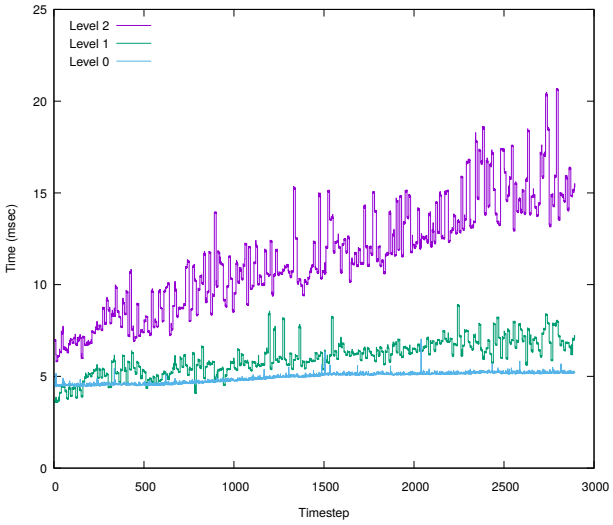

Figure 8. Runtime spent on different mesh refinement levels in CleverLeaf per timestep.

```
GROUP BY  kernel,mpi.function,mpi.rank
```

This aggregation scheme gives us the time spent in each computational kernel and each MPI function for each MPI rank. Figure 7 shows the time distribution across MPI ranks for total computation and communication/synchronization time, as well as the top two MPI functions and computational kernels in CleverLeaf. Overall, there is a small amount of imbalance in total computation time, and accordingly in MPI time as well. The imbalance in the top two computational kernels accounts for less than half of the total computational imbalance, which suggests that a significant part of the computational imbalance originates elsewhere. The `advec-mom` kernel shows almost no imbalance.

E. Studying AMR performance

So far, we have demonstrated simple aggregation schemes to study the performance of computational kernels, communication overhead and load balance. These examples are fairly straightforward, and similar functionality is available in many existing HPC performance analysis tools. In our next example, we demonstrate the distinct feature of our approach: customized aggregation schemes that let us distinguish user-defined, application-specific data dimensions in the performance data. We use this capability with CleverLeaf to analyze the behavior of the adaptive mesh refinement approach. To do so, we use Caliper's on-line aggregation service with an aggregation scheme that includes all annotation attributes in the aggregation key, including the main loop iteration number and the application-specific AMR level. The scheme records the visit count for each entry and aggregates the runtime:

```
AGGREGATE count,sum(time.duration)
GROUP BY  function,annotation,amr.level,\
  kernel,iteration#mainloop,\
  mpi.rank,mpi.function
```

Using this aggregation scheme, the CleverLeaf simulation run produced 257 592 profile records per process. Now, we can use off-line aggregation to address specific anaylsis questions. The different data dimensions in the profile allow us to study a wide range of performance aspects: by distinguishing main loop iterations, we can study performance development over time; likewise, the AMR level annotation lets us study the mesh refinement behavior.

One interesting aspect is the amount of time spent processing on different mesh refinement levels. CleverLeaf uses three refinement levels from 0 to 2, with 0 being the coarsest and 2 being the finest. As the simulation progresses, the adaptive mesh refinement algorithm automatically adjusts the discretization mesh resolution in each part of the physical domain, assigning finer levels to parts with high activity and coarser levels to parts with less activity. We can extract the runtime spent in each refinement level as the simulation progresses from our profile data by aggregating `time.duration` over AMR level and simulation timestep. As we are only interested in pure computation time, we exclude records with MPI functions from the aggregation. This is accomplished with the following aggregation scheme:

```
AGGREGATE sum(time.duration)
WHERE     not(mpi.function)
GROUP BY  amr.level,iteration#mainloop
```

Figure 8 shows the result. We can see that the runtime spent processing AMR level 0 stays almost constant over the entire simulation run. In contrast, the time spent processing level 1 increases slightly as the simulation progresses, and the time in level 2 increases significantly.

We can also examine how the time spent processing the different mesh refinement levels varies across MPI ranks (Figure 9):

```
AGGREGATE sum(time.duration)
WHERE     not(mpi.function)
GROUP BY  amr.level,mpi.rank
```

Here, we see that the runtime proportions spent in each refinement level are similar on most ranks, with some exceptions: for example, rank 8 spends more time in level 1 than 0, and rank 7 spends less time in level 0 than most other ranks.

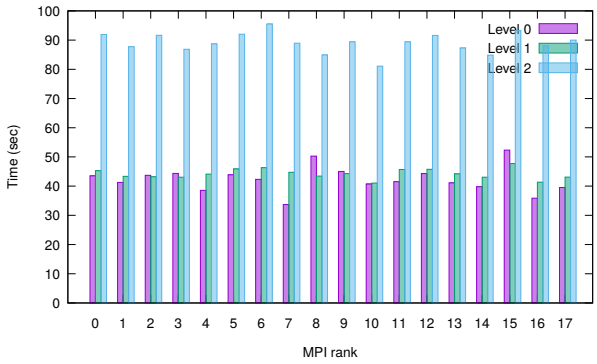

Figure 9. Runtime spent on different mesh refinement levels in CleverLeaf per MPI rank.

F. Discussion

In the CleverLeaf example, Caliper's flexible aggregation approach allowed us to address a wide variety of performance analysis questions with custom aggregation schemes and a combination of on-line and off-line data aggregation. All experiments ran with the same program executable using the same instrumentation annotations, we only changed the aggregation schemes to create a range of different performance profiles.

Also, note that the combination of on-line and off-line aggregation leaves multiple ways to obtain the same end result, letting us shift the bulk of the data aggregation from on-line to off-line processing and vice versa.

VII. Related work

The HPC community has created a wide range of commercial and open-source performance analysis tools for tracing or profiling parallel applications. Traces record individual program events over time, making them suitable for studying time-dependent behavior with detailed visual exploration tools. Typical examples are timeline displays, e.g. in Vampir [9] or ParaVer [10]. Scalasca [11] uses traces for automatic bottleneck detection. As a disadvantage, trace sizes grow linearly with the length of the execution, which usually limits tracing to short, dedicated performance analysis experiments.

Many performance tools use on-line aggregation to create performance profiles: mpiP [12] is a lightweight profiler focused on MPI communication; Score-P [13], TAU [14], Open|SpeedShop [15], and HPCToolkit [16] are full-featured profiling tools used in production that support typical HPC programming models (e.g., MPI and OpenMP) and the analysis of user code. TAU uses instrumentation-based (synchronous) data collection, whereas Open|SpeedShop and HPCToolkit focus on low-overhead data collection using asynchronous sampling. Like our approach in Caliper, Score-P supports both synchronous and asynchronous data collection.

The fundamental distinction of our approach are the flexible key:value data model and customizable aggregation schemes. In traditional tools, the aggregation key is determined by the profilers' implementation, often based on the underlying data models and file formats, and set up for one or few specific profile types, such as per-thread call-path profiles. Some tools can collect additional data dimensions, such as phase profiles in TAU [17] or time-series profiles in Score-P [18], which record separate call-path profiles for each main loop iteration to keep some dynamic information (i.e., performance variation across iterations) intact. Overall, however, the selection of aggregation schemes in the aforementioned tools remains limited, and adding new features such as time-series profiling can be a significant effort that may require file-format extensions as well as support in the supporting analysis toolchain. To our knowledge, our solution is the first on-line performance data aggregation approach to support fully customizable, user-defined aggregation schemes.

Many HPC performance analysis tools provide graphical user interfaces, e.g. the Open|SpeedShop GUI, hpcviewer for HPCToolkit, Cube for Score-P and Scalasca, and ParaProf and PerfExplorer [3] for TAU, that implement off-line data aggregation functionality for interactive exploration of performance profiles. However, their implementations also use hard-coded aggregation schemes that are limited by the underlying static data models. Cube provides a flexible language [4] to create new reduction operators and derive aggregation variables; however, aggregation keys cannot be modified. Moreover, the graphical report explorers are typically serial/single-node programs, which can become a performance bottleneck when analyzing large-scale experiments. While Caliper does not include a dedicated graphical user interface, we provide a highly scalable off-line query program for cross-process data aggregation.

VIII. Conclusions

Data aggregation mechanisms are widely employed in performance analysis tools for a wide range of applications, starting with data reduction at runtime using on-line event aggregation, to off-line analytical aggregation for extracting insights from previously collected data. However, aggregation solutions in traditional HPC-focused performance analysis tools are built around static, special-purpose data models and support only a few, pre-defined profile types, such as call-path profiles. Our approach lets users formulate custom, application-specific aggregation schemes, bringing a level of customizability previously only found in general-purpose database and data analytics frameworks into the performance analysis domain. Crucially, our aggregation model can be applied to scalable cross-process aggregation, off-line analytical aggregation, and low-overhead on-line event aggregation using the same description language.

In the CleverLeaf example, we have shown how customizable aggregation schemes let us tune profile data collection for specific performance analysis tasks, such as load balance or communication overhead studies. Moreover, the ability to include user-defined, application-specific data dimensions in aggregation schemes goes beyond the capabilities of traditional profilers: application-specific attributes, such as the AMR level in our example, let us study performance aspects that could previously not be obtained.

ACKNOWLEDGEMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 and supported by the Office of Science, Office of Advanced Scientific Computing Research as well as the Advanced Simulation and Computing (ASC) program.

LLNL release LLNL-CONF-xxxxxx

REFERENCES

[1] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Supercomputing 2003 (SC'03)*, Phoenix, AZ, November 15-21 2003.

[2] (2011) Component-based tool framework (CBTF). [Online]. Available: http://ft.ornl.gov/doku/cbtfw/start

[3] K. A. Huck and A. D. Malony, "PerfExplorer: A performance data mining framework for large-scale parallel computing," in *Supercomputing 2005 (SC'05)*, Seattle, WA, November 12-18 2005, p. 41.

[4] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, "Cube v4: From performance report explorer to performance analysis tool," *Procedia Computer Science*, vol. 51, pp. 1343–1352, Jun. 2015.

[5] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance introspection for hpc software stacks," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Computer Society, Nov. 2016, pp. 47:1–47:11, lLNL-CONF-699263. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014967

[6] A. Arsenlis, W. Cai, M. Tang, M. Rhee, T. Oppelstrup, G. Hommes, T. G. Pierce, and V. V. Bulatov, "Enabling strain hardening simulations with dislocation dynamics," *Modelling and Simulation in Materials Science and Engineering*, vol. 15, no. 6, 2007.

[7] B. T. N. Gunney, A. M. Wissink, and D. A. Hysom, " Parallel Clustering Algorithms for Structured AMR," *Journal of Parallel and Distributed Computing*, vol. 66, no. 11, pp. 1419–1430, 2006.

[8] S. Galera, P.-H. Maire, and J. Breil, "A two-dimensional unstructured cell-centered multi-material ALE scheme using VOF interface reconstruction," *Journal of Computational Physics*, vol. 229, 2010.

[9] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler, "Performance optimization for large scale computing: The scalable VAMPIR approach," in *Proceedings of the 2001 International Conference on Computational Science (ICCS 2001)*, San Francisco, CA, May 28-30 2001, pp. 751–760.

[10] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, 1995, pp. 17–31.

[11] 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, Apr. 2010. [Online]. Available: http://apps.fz-juelich.de/jsc-pubsystem/pub-webpages/general/get_attach.php?pubid=142

[12] J. Vetter and C. Chambreau, "mpiP: Lightweight, Scalable MPI Profiling," http://mpip.sourceforge.net.

[13] "Score-p user manual," https://silc.zih.tu-dresden.de/scorep-current/pdf/scorep.pdf, 2015.

[14] S. Shende and A. D. Malony, "The tau parallel performance system," *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.

[15] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open|speedshop: An open source infrastructure for parallel performance analysis," *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008.

[16] 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.

[17] A. D. Malony, S. Shende, and A. Morris, "Phase-based parallel performance profiling." in *PARCO*, 2005, pp. 203–210.

[18] Z. Szebenyi, F. Wolf, and B. J. N. Wylie, "Space-efficient time-series call-path profiling of parallel applications," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC09), Portland, OR, USA*. ACM, November 2009. [Online]. Available: http://dl.acm.org/ft_gateway.cfm?id=1654097&type=pdf&coll=DL&dl=GUIDE&CFID=126268637&CFTOKEN=35781611