Software-defined Events through PAPI

Anthony Danalis, Heike Jagode, Thomas Herault, Piotr Luszczek and Jack Dongarra
{adanalis | jagode | herault | luszczek | dongarra}@icl.utk.edu

Innovative Computing Laboratory
University of Tennessee
Knoxville, TN, USA

***Abstract*—The methodology and standardization layer provided by the PAPI performance monitoring library has played a vital role in application profiling for over a decade. It has enabled sophisticated performance analysis tool designers, and performance conscious scientists to gain insights into their applications by simply instrumenting their code using a handful of PAPI functions that "just work" across different hardware components.**

In the past, PAPI development had focused primarily on hardware-specific performance metrics. However, the rapidly increasing complexity of software infrastructure raises new measurement and analysis challenges for the developers of large-scale applications. In particular, acquiring information regarding the behavior of libraries and runtimes—used by scientific applications—requires low-level binary instrumentation, or APIs specific to each library and runtime. No uniform API for monitoring events that originate from inside the software stack has emerged.

In this paper, we present our efforts to extend PAPI's role so that it becomes the de-facto standard for exposing performance-critical events, which we refer to as Software-defined Events (SDEs), from different software layers. Upgrading PAPI with SDEs enables monitoring of both types of performance events—hardware- and software-related events—in a uniform way, through the same consistent PAPI interface. The goal of this paper is threefold. First, we motivate the need for SDEs and describe our design decisions regarding the functionality we offer through PAPIs new SDE interface. Second, we illustrate how those events can be utilized by different software packages, specifically, by showcasing their use in the task-based runtime PaRSEC, and the HPCG supercomputing benchmark. Third, we provide a thorough performance analysis of the overhead that results from monitoring different types of SDEs and discuss the trade-offs between overhead and functionality.

***Index Terms*—PAPI, SDE, Software-defined Events, libraries, runtimes, instrumentation, performance, PaRSEC, HPCG**

I. Introduction

Developing applications using some form of a modular, or layered design—where different logical operations are performed by different, smaller units of a large, complex application—is not only a good software engineering principle, but is also common practice across diverse fields. Focusing on the field of High Performance Computing (HPC), the community has moved away from the large monolithic FORTRAN codes that dominated the field a few decades ago, and has adopted more structured designs which foster collaboration between groups of people, and code reuse. Besides adhering to good software engineering principles, this transition was necessitated by the increasing complexity of hardware platforms, which transitioned from single node machines, to distributed memory heterogeneous supercomputers. As a result, many modern HPC applications are not only internally organized in smaller units, but also use external libraries for functions such as communication and synchronization (with MPI being the leading choice), runtimes for on-node parallelism (such as OpenMP), and a plethora of external libraries for functions such as math, or access to accelerators. In the rest of this document, we will discuss performance aspects of such applications, and we will use the term "module" for any code entity, such as library, runtime, class, etc., which can be used as a building block of a larger application.

In HPC, where application performance is critical, there is a drawback to adopting a design that is non-monolithic. Specifically, the developers of one module lack information regarding the internal behavior of modules which they are using, but they did not develop themselves. For example, when using a communication library such as MPI, the application developer does not know if the actual data transfer of a non-blocking call took place after `MPI_Wait()` was called, or earlier. Similarly, when using a task execution runtime, the developer does not know how many tasks are available for execution at any given time. In summary, when complex applications are properly structured in multiple modules, then lack of information exchange between different modules can lead to sub-optimal interaction between different modules, which can lead to loss of performance.

Some projects with wide adoption, such as MPI and OpenMP, have been developing a "tools' interface" (MPI_T [1] and OMPT [2] respectively). This is an effort to create custom hooks inside the libraries which implement these standards, such that external performance analysis tools can use these hooks to extract information about the internal behavior of these libraries. These efforts offer a solution to the problem of exchanging information between modules, without breaking the modularity of complex applications. However, developing library specific APIs is not a scalable approach. It is not feasible for every library developer group to establish their own API and expect that application developers, or performance analysis toolkits will adopt them all. As a solution

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Additionally, some of this material is based upon work supported in part by the National Science Foundation NSF under grant 1642440 "SI2-SSE: PAPI Unifying Layer for Software-Defined Events (PULSE)".

to this problem, we have developed an API for exporting software-defined events (SDEs) through PAPI. Being the defacto standard middleware layer for hardware performance events, with wide adoption by toolkits and application codes, and wide availability on most software stacks, PAPI is the perfect vehicle for supporting and delivering generic SDEs.

In the rest of this paper, we discuss the high level decisions we have made in the design of PAPI SDEs, we describe example uses which illustrate the usefulness of SDEs in modern libraries, and discuss the performance effect of adding PAPI SDEs in HPC libraries.

II. Design and Functionality

The main functionality of PAPI Software-defined Events is that of revealing internal information of black-box modules of the software stack to other modules, or performance analysis toolkits. PAPI has a near ubiquitous presense on modern systems, and wide support by performance analysis tools. Consequently, if the developers of a library use PAPI SDEs to export internal information about their library to the outside world, they can expect this information to be used for more sophisticated performance analysis of applications which use the library. Furthermore, developers of applications which use multiple external libraries, can use SDEs found in these libraries to understand and fix performance issues due to poor coordination between the different libraries used in their applications.

The developers of a given software module are experts in both the semantics of their module and its implementation. For this reason, we designed our SDE functionality so the experts can export whatever information they deem important, and wherever they think it should be exported from. We do not attempt to dictate the type of information which will be exported as an SDE. We simply offer the mechanisms and middleware layer for any kind of information to be exported.

For example, a task scheduling runtime could export the number of available tasks at different points in time, but a climate code could export a performance metric of emulated-years per second. PAPI SDEs were not designed for a performance expert to annotate a third party code so they can measure how long it took for a code segment to execute. There are other tools for this type of annotation. Rather, PAPI SDEs are meant to be used by the developers who write a software module so that the internal behavior of their module can be understood better by those that use it.

A. Design decisions

Since the early design and development stages of PAPI SDEs, we interacted heavily with members of the performance analysis toolkit community, as well as developers of libraries and runtimes which would make natural targets for early adoption. These interactions revealed two principal concerns.

- Performance analysis toolkit communities strongly emphasized the importance of **preserving the existing API** which is currently exported by PAPI for measuring hardware events.
- Library and runtime communities were mostly concerned with the **performance overhead** caused by the introduction of SDEs in their codes.

Since our project is positioned as a middleware layer, success depends on adoption by other software modules and toolkits. As a result, we used the concerns raised by the community to guide our design decisions.

To address the requests made by developers of toolkits, we implemented SDE support in PAPI as a new component which provides an API for registering SDEs into the existing PAPI framework. After SDEs have been registered, they can be accessed using the same API which has always been used for accessing the hardware events that PAPI supports. In other words, we created a new API for library writers to register their software events with PAPI, but we maintained the existing API (e.g. `PAPI_start()`, `PAPI_read()`, `PAPI_stop()`) for users and toolkits to monitor these events.

Satisfying the performance concerns of library developers was more challenging, since there is often a tradeoff between overhead and functionality. Our approach to handling the tradeoff is multi-faceted and enables the library developers to make the choices that best fit the requirements of their code. Specifically, we provide several types of SDEs that strike a different balance between overhead and functionality. We discuss these SDE types in the following sections.

B. PAPI SDE counter types

1) **Registered counters** offer developers the ability to register an existing internal variable of their library as a PAPI counter for an SDE. The registration of such a counter happens only once, presumably during the initialization of the library, so the performance overhead due to the registration is constant and negligible. Through this SDE type, PAPI enables entities outside a library to read a variable that already existed in the library and was updated by library code when needed. Therefore, no additional work needs to be performed by the library code in order to support this type of SDE.
2) **Registered function pointers** offer library developers the ability to register a special-purpose function: internal to the library and invoked by PAPI. When called, the function acts as an accessor to deliver the counter value. This type of SDE is useful when a library has no pre-existing variable that acts as a counter for an event. Also, in case the value of the event counter must be derived from a complex internal state of the library rather than a single variable. The registration of a function pointer happens only once, so the performance overhead of the registration is constant and negligible. For this type of SDE, the registered function is only used to read a value, not to update a counter and therefore it does not have to be called by the library. Instead, it is only called by PAPI when the application which uses the library makes a call to `PAPI_read()`. Therefore, this type of SDE does not add overhead to the *fast path* of the library. In other words, when the library is involved in a "maximum

performance run," where PAPI is not used to monitor its behavior, the registered function does not get called.

3) **Created counters** offer library developers the flexibility of creating a counter inside PAPI, instead of having a counter inside the library. The creation of such a counter happens only once, but updating the value of a created counter requires a call to a PAPI SDE function. This type of SDE has two main benefits over registered counters. First, created counters are always thread safe. As a result, using a created counter relieves the library developers from the need to use explicit thread safe code every time they update the counter. Second, since PAPI is aware of every update of the counter value, this type of SDE lends itself to more accurate overflow support, or other type of notification of performance analysis toolkits. The drawback of this type of SDE is that it requires a call to a PAPI SDE function inside the library code every time the event counter needs to be updated, and therefore it has higher overhead than registered counters.
4) **Recorders** offer library developers the ability to record a series of values associated with an event. Similarly to created counters, the memory associated with a recorder is managed internally by PAPI, and the creation of a recorder happens only once, but a PAPI SDE function needs to be called for every new value that is being recorded. PAPI makes no assumptions about the type of the variable being recorded. Instead, the API requires only a pointer to the variable, and the size of the variable. This way, library developers are free to record any type of data from simple integers to complex structures, or even strings, or arrays of values.
   When a recorded is created, PAPI automatically creates a few additional auxiliary counters. The first has the same name as the recorder with the additional suffix "`:CNT`". This counter holds the count of elements that have been recorded at any given time. In addition, there are five more auxiliary counters automatically created for each recorder. When read, these counters return the quantiles of the recorded distribution, and in particular the minimum and maximum values recorded, as well as the three quartiles (i.e., 25%, median, 75%). The names of these counters are formed by adding one of the prefixes {`:MIN`, `:Q1`, `:MED`, `:Q3`, `:MAX`} to the name of the recorder. In contrast with the auxiliary counter "`:CNT`", these statistical counters are optional and depend on the ability of PAPI to compare the recorded values. Since the recorded values can be of arbitrary type, when a library creates a recorder it is expected to provide a pointer to a function which is able to compare two values of the type that is recorded. If the function pointer is `NULL`, then the statistical counters are not created.
5) **Groups** offer library developers the possibility of aggregating the values of multiple counters into a single entity. Groups are implemented as first class citizens and can be added into larger groups recursively. When a library creates a group it must specify if the value that is reported when this group counter is read consists of the minimum, the maximum, or the sum of the values of the counters which belong to the group. Both registered and created counters can be added to groups, but not recorders (however, the auxiliary counters associated with a recorder could be added to groups). In terms of performance overhead, groups do not require any additional code to be inserted in the fast path of a library. The value of a group is assembled when a user application calls `PAPI_read()` by reading the values of all the counters which belong to the group.

C. Overhead-Functionality tradeoff

The design we have adopted gives full control of the overhead-functionality tradeoff to the library developers. Each group can choose if the functionality provided by a feature justifies the amount of performance overhead this features will add to their library, or if they want to limit the SDE types they will utilize to those with zero overhead.

Libraries that already count internal quantities and events, but do not have a standardize way to export this information to the outside world, will benefit from PAPI SDEs while facing zero performance overhead. Libraries with no event counting functionality can benefit by adding internal counters, or accessor functions, which as we will further discuss in section IV, can still be done with negligible or zero performance overhead. Libraries with events which do not occur frequently enough for performance overhead to be of primary concern can use created counters to communicate these events to users, or analysis toolkits that are "listening". And finally, libraries with events whose evolution over time is important can record long series of custom event values for advanced analysis by performance conscious users and sophisticated toolkits.

III. Examples

A. PaRSEC

1) Overview: PaRSEC is a runtime environment that supports the execution of task systems on large scale distributed and hybrid computers [3]. Applications written in PaRSEC express their parallelism as a set of sequential tasks which exchange data following a directed acyclic graph (DAG). The PaRSEC runtime fulfills two roles. First, it executes the application tasks on the computing nodes and accelerators using a complex set of schedulers guided by the application developer, by the progress of the execution, and by the hardware capabilities. Second, it moves data between the computing resources, transparently for the programmer.

The PaRSEC engine is at the heart of the runtime environment: it provides schedulers, data management capabilities, and various hardware support. Task systems are exposed to the engine through Domain Specific Languages (*e.g.*, Parameterized Task Graphs [4], Dynamic Tasks Discovery [5], *etc.*). The PaRSEC environment is completed with debugging and

tracing tools that enable developers and users to visualize the DAG of tasks, and the execution and scheduling decisions.

2) PaRSEC schedulers: The main advantage of a task system is that it does not enforce an order on the execution of tasks, but exposes parallelism to allow dynamic decisions. This is important, because in large scale computing, where the hardware increases in complexity (as with hybrid computers providing most of their computing power from accelerators), the best order of execution cannot be entirely planned in advance, and part of the scheduling decisions must remain dynamic.

However, depending on the application and the platform, different scheduling approaches achieve different performance. PaRSEC is built on top of the Modular Component Architecture (MCA), that was developed for Open MPI [6]. MCA provides a tool to expose a public API (a component) that can be implemented with many algorithms (the modules), and lets the end-user or the programmer select at run time which module will be used to provide the service. Schedulers in PaRSEC are implemented over an MCA component, allowing the user to load and unload different schedulers to guide the execution of different DAGs of tasks on different hardware, even within the same application.

As the selection of a scheduler is a performance-critical task, information on the status of the scheduler and the reasons it takes some decisions during the execution are valuable feedback for the users and developers. To expose this information in the most straightforward way, we introduced PAPI SDE counters in the PaRSEC schedulers, allowing users to exploit the standard performance tools to explore these counters and experiment with different schedulers.

3) Task-queues for different schedulers: There are 10 different schedulers in the PaRSEC base distribution. We will not describe them exhaustively here, but similar principles apply to all of them. Depending on the scheduler, a varying number of queues exist: the simplest scheduler uses a global queue to order all tasks as they become ready in a first-in first-out approach. For this scheduler, it is simple to count the number of tasks ready to be executed at a given time: it is the length of the queue.

Reporting the number of ready tasks is more complex for more hierarchical schedulers. The Local Flat Queue scheduler, which is the default in PaRSEC, uses a hierarchy of bounded queues: each thread is bound to a single core and maintains its own queue of ready tasks. When a thread generates more work (because a task it completed releases more tasks), the new tasks are inserted into the bounded queue of the thread. When the queue is full, remaining tasks are queued in a shared queue. When selecting a task for execution, a thread will first try to pop it from its local bounded queue. If that queue is empty, the thread will try to steal a task from the thread bound on the core closest to its own core, on the memory hierarchy of the machine. The thread tries to steal from other cores of the same socket, and if none are ready, it will steal from the main shared (and unbounded) queue. If that queue is empty, it will then try all the other cores on the node, and cycle until new work is found or the operation is completed.

PaRSEC aims at reducing the scheduling overhead. All these shared queues are implemented with lock-free algorithms, and a minimal number of atomic operations. Because they are shared, instrumenting them to provide a counter of length would increase the number of atomic operations and thus the scheduling overhead. To avoid this, we opted for a more scalable approach: each thread maintains a collection of thread-specific counters; when a thread inserts a task in a queue, or removes a ready task from a queue, it updates the corresponding local counter.

Since all threads can steal work from each queue, the sum of the counters across all threads is needed to compute the number of ready tasks available in each queue. Still, that number is not exact because no synchronization ensures that an element is not being inserter, or removed while the sum is computed. However, this is the trade-off chosen by PaRSEC developers to provide approximate information about the ready-task queues with a minimal impact on the normal execution of the schedulers.

Given the need to sum the local counters across threads, PaRSEC uses the registered function pointer SDE type, and provides accessor functions which perform the summation. Moreover, the bounded queues are by nature hierarchical: the union of the queues over the same socket represents the work that has a high probability to run on this socket, and an unbalance in the work leaning toward a socket versus another can lead to NUMA effects that can explain observed performance. Thus, we organized all available information by using the PAPI SDE grouping concept. The base counters expose the number of ready tasks at each level of the hierarchy of queues, and the groups are defined to aggregate these numbers following the hardware hierarchy. This way, we expose to the user more information about how many tasks are waiting, and where they might be executed. At the top level of the hierarchy, the largest group offers the user the simplest number: the number of tasks which are ready to execute and waiting in a queue at a given time, for the entire node. Figure 1 shows the evolution of multiple different PaRSEC queues under a typical workload.

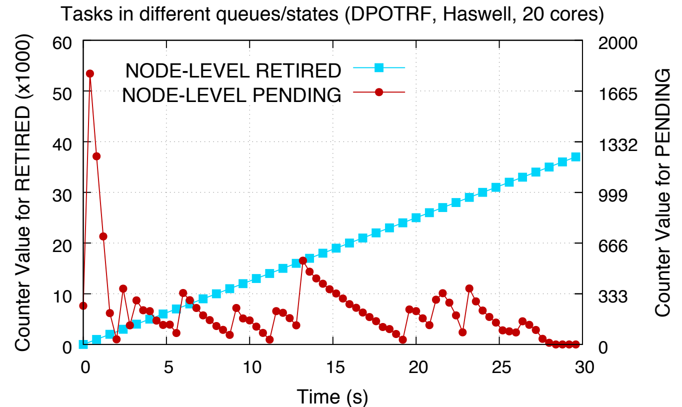

Fig. 1. PaRSEC queue length evolution over time.

B. High Performance Conjugate Gradients (HPCG)

1) HPCG Background and Relevance of SDE In Benchmarking Process: High Performance Conjugate Gradients [7] (HPCG) has been a benchmark for measuring performance of supercomputers since early 2014, and its design dates back to the year prior [8]. Since then, the biannual release of the HPCG performance list [9] complements some of the other HPC industry benchmarks such as TOP500 and Green500. While the same basic algorithm [10] was used for the HPCG implementation, over time, the later versions incorporated some elements of Multi Grid [11]. At the same time, using latency-hiding techniques and its algorithmic variants such as pipeling [12] is prohibited. This creates a dynamic interaction between the HPCG implementers that optimize the code and the committee that verifies the submitted results. Software-Defined Events play an important role by giving access to numerically significant events occurring in the implementation that may be monitored externally without accessing often proprietary details of the implementation.

2) HPCG Vendor Implementations: The vendor implementations of HPCG do not need to be disclosed in source code form except for extraordinary circumstances. This allows the vendors to maintain the control of how the information about low-level hardware details is disseminated, while providing the benefits of the platform-specific optimizations. This is almost identical situation of how BLAS [13]–[17] and LAPACK [18], [19] provide a reference implementation as open source code and the hardware-specific implementations are proprietary and mostly delivered in closed-source form as a binary-only libraries. This however, is where the analogy between linear algebra libraries (like BLAS and LAPACK) and linear algebra benchmark (like HPCG) ends. In order to gain large user base, the vendor libraries have an incentive to be both accurate and fast. In particular, they should be as accurate as the reference implementation and as fast as the hardware permits. On the other hand, the only incentive for the vendor implementation of a benchmark is to remain fast. This may potentially sacrifice the accuracy. Hence, HPCG officially mandates and enforces certain degree of accuracy inside the benchmark's execution harness. But in addition to the verification and validation (V&V) module of HPCG, checking for sufficient accuracy may be performed through mandatory software events. SDEs need to be embedded in the proprietary code to allow for monitoring numerical events in the solver that the multigrid and Krylov subspace iteration would normally generate. Also, these events may be used for the alternative input data that is not used for runs that report benchmark's official performance – the may be called verification data sets. Then, by manipulating the numerical properties of the linear system and observing the effects on the type and count of numerical events being reported, the closed-source implementation may be checked for the necessary accuracy. This is not only a tool for the HPCG committee that verifies the benchmark but also for the vendor's performance engineer that needs to know when a given optimization caused a numerical bug that lowered the achieved accuracy.

Let's consider one possible recorder SDE that may result from the main system solver in HPCG resulting from the following discretization of a *partial differential equation* (PDE) – a single degree of freedom heat diffusion with zero Dirichlet boundary condition:

$$Au = f \tag{1}$$

where A defines the discretization operator (a 3D regular 27-point stencil in case of HPCG). As in any iterative solver, computing the residual at each iteration i:

$$d^{(i)} = Au^{(i)} - f \tag{2}$$

helps in indicating convergence or progressing in the future iterations. As one of the monitoring metrics, we set up a recorder SDE for $\left\|d^{(i)}\right\|$ which allows us to ascertain that the right number of iterations were performed and that the convergence rate conforms to what is predicted by the theory.

The reference code computes $d^{(1)}, \ldots, d^{(50)}$ and uses $\left\|d^{(50)}\right\|$ as the reference by which the optimized code is evaluated. With the recorder SDE for each iteration, it is possible to monitor the optimized implementation much more closely. It is also possible to provide different discretization operator and observe if the convergence history would change accordingly. These and other evaluations are possible without the vendor disclosing the code of their implementation.

IV. Performance Overhead

In this section we provide an experimental evaluation of the performance overhead associated with SDEs. We offer a multi-faceted investigation, and we discuss our experimental methodology because accurate attribution of overhead is not always straight forward.

A. Benchmarks

The simplest form of measurements comes from benchmarking. All the experiments mentioned in this section were performed on a Haswell E5-2650 v3 with a frequency of 2.3GHz, running Linux with kernel version 3.10.0-514.26.1.el7.x86_64. The benchmarks were written in C (as is PAPI), and compiled with gcc 4.8.5 using optimization level "`-O3`".

For our first experiment our code invoked the PAPI SDE function which creates a counter (SDE type 3, in section II-B) and then used the SDE API to increment the counter over 100K times. Every time our benchmark called the function to increment the counter it also measured the time it took to execute this function by reading the CPU *time-stamp counter* using the x86 instruction `rdtsc`.

The results of this experiment are shown on the left side of Figure 2, in the form of a (light blue) violin plot overlapped with a box-and-whiskers plot. As can be seen in the graph, the median execution time of the function which increments a created counter was 14.3*ns*, the box (which includes 50% of the measurements) extend from 13*ns* to 17.8*ns*, and the

whiskers (which include 99% of the measurements) extend from around 9ns to around 21ns.

For our second experiment our code invoked the PAPI SDE function which creates a recorder (SDE type 4, in section II-B) and then used the SDE API to record over 100K values of type "`double`" into that recorder. Every time our benchmark called the function to record a value it also measured the time it took to execute this function (using `rdtsc`).

The results of the recording experiment are shown on the right side of Figure 2, in the form of a (green) violin plot overlapped with a box-and-whiskers plot. As can be seen in the graph, the median execution time of the function which records a value was 17.4ns, the box extend from 16ns to 17.4ns, and the whiskers extend from around 12ns to around 44ns.

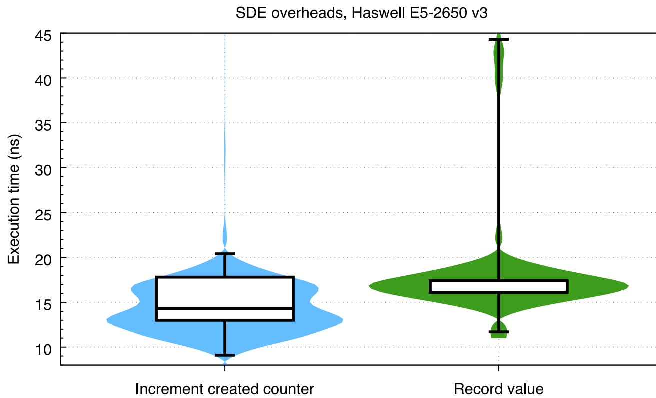

Fig. 2. Execution time measurements of SDE functions.

The violin plots of both experiments give a more accurate picture of the distribution of measurements and they reveal some outliers. In the case of the created counter there is a small clump of values around the $30ns \div 35ns$ range, but they constitute an insignificantly small fraction of the data set. In the case of the recorder there is a more significant clump of values around 40ns. Also, the recorder has some outliers that are in the range of $\approx 4\mu s$ (not shown in this graph). These, although they are rare enough not to affect the quartiles of the distribution, they are not due to noise, but rather they are an implementation artifact. Specifically, we have implemented recorders using contiguous memory, which is allocated in 4KiB increments, to avoid having a large memory overhead in libraries were recording will not be heavily used. As a result of this policy, for every 512 values of type `double` that are recorded, there will be a call to `realloc()` in order to increase the available space. The parameter which defines the size of the increment of memory allocation can be tuned to reduce the occurrence of reallocating (and copying) memory, and the overhead associated with it. However, as we demonstrate in this paper, the overhead is already insignificant, so fine tuning the increment size should not be a concern for most users.

The other three types of SDEs mentioned in II-B, namely *registered counters*, *registered function pointers*, and *groups*, do not have an inherent overhead, and so they cannot be directly measured with benchmarks in a systematic way. Registering a library variable (or a function) as an SDE counter does not alter the execution of the library code, and therefore it has zero impact on performance. The same is true if a library chooses to organize a set of registered counters as a group. Clearly, if a user application calls `PAPI_read()` to read the value of an SDE there will be an overhead in the execution of the application, due to the reading, but this overhead does not come from the library, but rather from the call to `PAPI_read()`. If an application uses a library which contains the "zero-overhead" SDE types mentioned above, but the application never calls PAPI functions to read the values of the counters, then the application will not experience any overhead due to the existence of the SDEs in the library.

There is a scenario in which registered counters can cause overhead to a library code. Namely, when a library wishes to export information about an internal event, but does not already contain any code to record this event. For example, consider the case of a communication library where the developers wish to create an SDE which will report the number of bytes transferred over the network. If the library did not previously record that number, it will now need to be modified so that a variable holding the number of bytes transferred is incremented every time a data transfer takes place. If, in addition, we consider that the library is multithreaded, then we see the need for this variable to be atomically incremented, or protected by some thread safe critical section code. Clearly, such a modification would cause a performance overhead to the library, and that overhead would be present even if the user application never calls `PAPI_read()`.

However, it can be argued that this overhead is really due to the additional functionality (keeping track of a value which was not tracked before) and not due to the fact that this value is exported as a PAPI SDE. Regardless of the semantics of overhead attribution, such overheads are highly dependent on the specifics of the library code and can not be measured by a benchmark in a meaningful way.

In the following sections we provide a performance overhead analysis of different types of SDEs across a series of more realistic uses within existing, well known, third-party libraries.

B. ATLAS

In the previous section we quantified the overhead due to calling an SDE function. However, in addition to the actual code this function executes in order to update a counter, there is also an indirect cost due to cache pollution. Namely, the implementation of all types of SDEs relies on some meta-data structures that are used internally by PAPI for book-keeping. Furthermore, the *recorder* SDE continuously stores new data in memory, and when need be it allocates more memory and performs a copy (due to `realloc()`).

In this section, we perform a study which aims to quantify the total overhead caused by the existence of SDEs in a time critical library. For this purpose, we used the dense linear algebra library *ATLAS* (Automatically Tuned Linear Algebra

Software). The rational was that upon installation ATLAS tunes its kernels to maximize the utilization of all hardware features of the system in which it is being installed, and this is especially true for the cache hierarchy. Therefore, inserting an SDE inside an ATLAS kernel, after the tuning has taken place, should interfere with the tuned kernel and provide a worst case scenario for the SDE overhead.

In graphs 3 and 4 we present the results of our experiments. Both graphs show distributions of runs, because for each set of parameters we made over 1000 runs and we plot the whole distribution (as well as a boxplot). Both graphs show results from the same type of experiment, the only difference being the problem size varying between small and medium (matrix size 504x504 for the first, and 2016x2016 for the second). The kernel we chose for all experiments was `ATL_dNBmm_b0.c` which performs a double precision matrix-matrix multiply on matrix tiles of size 56x56. The ATLAS version we used was 3.10.3, and all our experiment were performed on the same hardware as the benchmarks described in section IV-A (Haswell E5-2650 v3 @2.3GHz).

The code snippet below is taken from this kernel and shows only the part which we modified for our experiment.

```
void ATL_USERMM(...){
   ...
   do /* N-loop */
   {
      do /* M-loop */
      {
         rC0_0 = _mm256_setzero_pd();
         rC1_0 = _mm256_setzero_pd();
    ...
```

The first (dark purple) violin in our graphs shows the distribution of execution times for the vanilla version of the ATLAS kernel. This version does not contain any modifications done by us. The second (orange) violin shows the execution time when the outer loop (N-loop) of the kernel is modified to add a call to the SDE function which increments a created counter. The Y2 axis of the graph (the one on the right) shows the percent overhead in comparison to the median execution time of the vanilla version. As can be seen in the graph the total overhead on the kernel, after adding the SDE call, was about 1% for the small size problem and less than 0.5% for the larger problem—and this is true for all quartiles compared to the corresponding quartiles of the vanilla distribution. The number of iterations of the N-loop was 2,268 for the small problem size and 36,288 for the large one, so the total execution overhead of incrementing the SDE counter was 52*ns* per increment for the smaller problem and 62*ns* per increment for the larger problem (comparing the medians of the distributions).

The third (blue) violin shows the execution time of the ATLAS kernel when a call to the SDE function which increments a created counter is inserted to both loops. In this case the total iteration count jumps to 34,020 for the smaller problem and 544,320 for the larger. The overhead becomes about 3% and 1% for the two problem sizes, but the overhead per increment drops to 12*ns* and 14*ns* respectively.

The fourth (magenta) violin shows the execution time of the kernel when a call the the SDE function which records a value inserted to the outer loop. The overhead of this SDE was about 1% for the smaller problem and less than 0.5% for the larger problem, which translates to 56*ns* per recording and 37*ns* per recording, respectively.

Finally, the fifth (green) violin shows the execution time of the kernel when a we record a value in both loops. The overhead in this case was about 4% for the smaller problem and 2% for the larger problem, which translates to 18*ns* per recording and 32*ns* per recording, respectively.

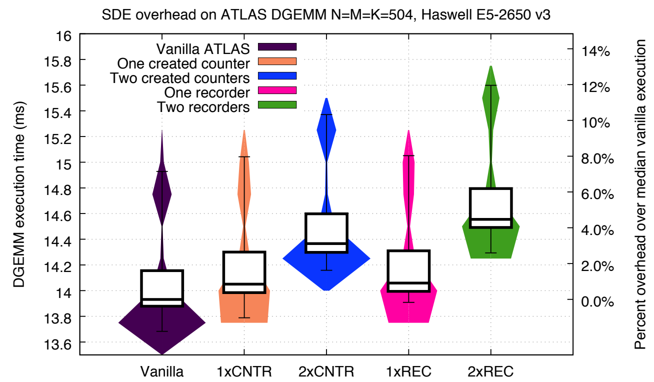

Fig. 3. ATLAS with small size matrix.

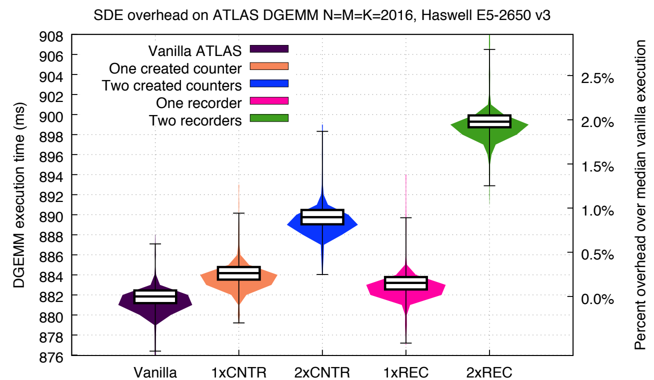

Fig. 4. ATLAS with medium size matrix.

Clearly, the examples we used in this section are extreme and unreasonable. We do not envision performance critical libraries, such as ATLAS, to add hundreds of thousands of invocations to our API inside their kernels. However, even under this level of load, we see that the performance overhead remains within a few percent. In the following sections we demonstrate the performance overhead in more realistic use scenarios.

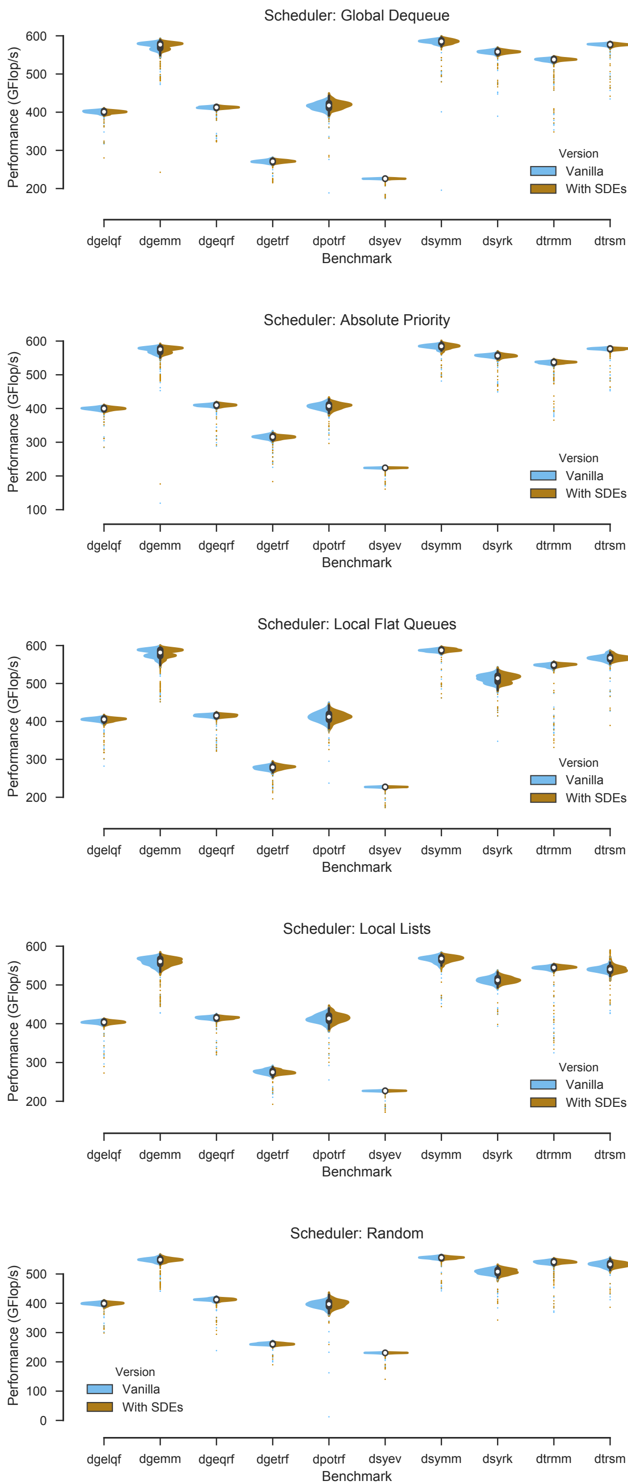

Fig. 5. Performance comparison between PaRSEC with SDE and ParSEC Vanilla, for different benchmarks and different schedulers. Matrix size of $4,000 \times 4,000$, on a Haswell E5-2650 v3 at 2.3GHz

C. PaRSEC

Figure 5 compares the performance of 10 different benchmarks running on top of PaRSEC, with 5 different schedulers, on a 20-core Haswell E5-2650 v3 at 2.3GHz, with and without SDE support. Violin plots with outliers are shown in order to focus on the impact of the SDE instrumentation on performance. As explained in Section III-A, the different schedulers use different hierarchies of lists, each of them being instrumented with PAPI-SDE counters that are aggregated using the PAPI-SDE function and grouping interfaces. The scheduler "Local Lists" features one list per core, while the scheduler "Local Flat Queues" adds a shared dequeue to those; "Global Dequeue" uses a single list modified only with atomics, while "Absolute Priority" uses a sorted list protected by a lock. The "Random" scheduler uses an array with read-write lock for resizing, and atomic swap for insertion and selection.

Names dgelqf, dgemm, dgeqrf, dgetrf, dpotrf, dsyev, dsymm, dsyrk, dtrmm and dtrsm are dense linear algebra operations from the DPLASMA library [20], which is written on top of PaRSEC. We chose the input size for the benchmarks such that it results in thousands of tasks, keeping each core relatively busy (depending on the DAG of tasks resulting from the operation). The violin plot shows the performance distribution over a sample of a thousand runs for each combination of parameters. Runs that were outside of twice the interquartile range have been sorted as outliers (shown as points on the figure), and they represent less than 5% of the total runs.

The figure shows that the SDE instrumentation has no statistically significant impact on the performance of the runs, for all benchmarks, and independently of the scheduler. This is the sought-after behavior for such instrumentation.

D. HPCG

In section III-B, we gave an example of a recorder SDE for 64-bit floating-point values that registers norms of the residuals $\left\|d^{(i)}\right\|$ in an iterative solver of an instrumented HPCG implementation. We set up additional recorder SDEs to more comprehensively address the complexity of the HPCG solver that uses Krylov space iterations with a multigrid preconditioner that features Gauss-Seidel smoother. For verification purposes, the extra counters recorded useful runtime information such as data sizes of neighborhood collectives' and smoothing errors at all three multigrid levels. In total, the standard 50-iteration run recorded over 1200 events per MPI process. To evaluate the overhead associated with such monitoring, we ran HPCG on a 65 node Inifiniband cluster with single-socket Intel Xeon x5660 CPU running at 2.8 GHz with 12 cores and MPICH 3.2.1. Figure 6 shows violin plots of performance for running both the reference and instrumented versions of HPCG on node counts from 1 to 65 (12 core to 780 cores). The scaling is nearly linear but it trails off at high core counts due to the global reductions inside HPCG that were designed to stress the interconnect – both versions show the same scaling behavior regardless of overhead. More importantly, the overhead due to PAPI SDE instrumentation has statistically indistinguishable

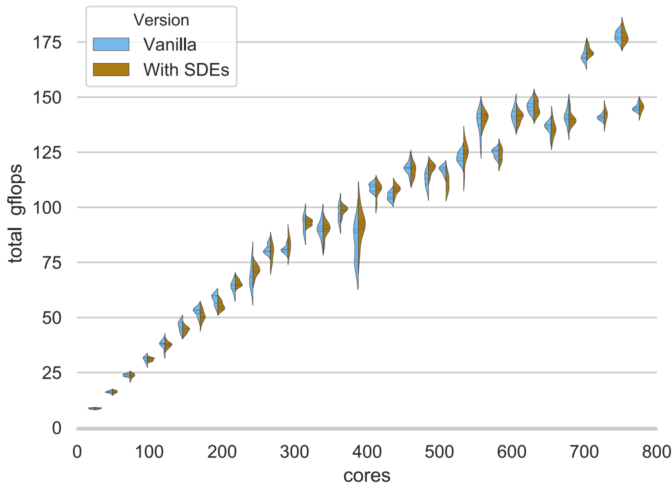

Fig. 6. Total performance for weak scaling of HPCG on a cluster with $65 \times 12 = 780$ cores.

effect on the resulting performance and the observed statistical distribution of performance samples. This is the desirable property for performance-critical benchmark runs.

Figure 7 shows similar information to the previous figure but it features performance per core as the core count increases from 1 to 65. Consequently, the figure shows the efficiency expressed as per-core Gflop/s rather than normalized percentage value. As mentioned already, the scaling of the interconnect does not keep up with the number of cores and, as a consequence, the per-core performance drops with the increasing number of cores. The overhead of instrumentation does not meaningfully change the distribution of the observed results which was already observed on the previous figure with the weak scaling results.

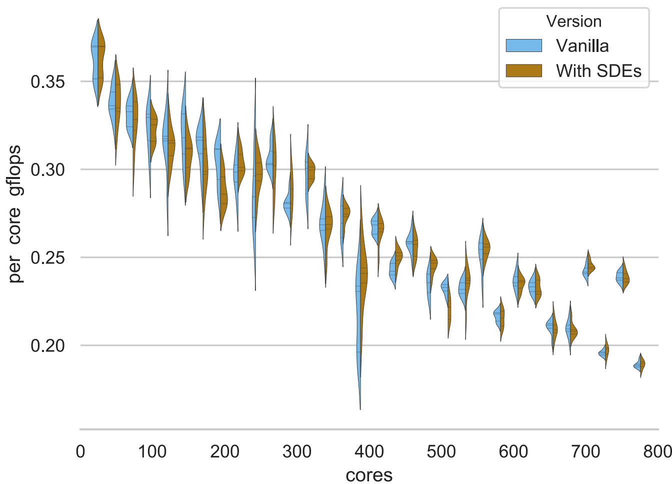

Fig. 7. Per-core performance for weak scaling of HPCG on a cluster with $65 \times 12 = 780$ cores.

V. Related Work

The need for software developers to acquire knowledge of the internal behavior of libraries has been recognized by some of the communities that develop performance critical libraries. Particularly, the de facto standard for developing distributed memory applications, MPI [21], and one of the leading efforts for delivering multi-threaded shared memory applications, OpenMP [22], provide both instrumentation and profiling mechanisms as part of their standard. The two distinct efforts, MPI_T [1] and OMPT [2], [23] respectively, make it clear once more that experts in performance critical libraries recognize the need for exporting internal library information to their users through instrumentation and profiling interfaces.

- MPI_T is an interface for tools introduced in the 3.0 version of MPI. It allows tools to understand and manipulate internal MPI variables in order to provide a more efficient and application-adapted execution environment. Similar to the PAPI interface, the MPI Tool Interface allows the implementation to specify internal control and performance variables, allowing tools to iterate over all possible variables to query their properties, retrieve descriptions about their meaning, and access and (if appropriate) alter their values.
- The OpenMP standard includes OMPT, a first-party interface for performance tools. It offers functions to query OpenMP states and callback functionality for relevant OpenMP events. This allows tools to explore details of an OpenMP implementation, examine runtime states associated with an OpenMP thread, identify parallel regions and tasks, and to collect call stacks.

While these efforts provide a useful view of the execution of a parallel application, the granularity of the analysis interval is too coarse grain (mostly at the level of entry and exit point of MPI functions, OpenMP regions or tasks). More importantly, unlike the approach described in this paper, these solutions are specific to MPI and OpenMP, and so, they do not fit easily or naturally into the performance tool ecosystem. To incorporate them, developers of performance critical applications or higher level profiling tools would have to implement profiling code customized for the communication layer of their parallel application. This paper addresses these challenges. The new SDE support in PAPI is not limited to a specific library, but enables *any* library developers to expose internal information about their libraries in a *consistent and standardized* way. Additionally, the PAPI SDE extension enables performance toolkits and application developers to capture and utilize such information across all the software layers used in an application.

TAU [24] is a profiling and tracing toolkit aimed at the performance evaluation of parallel programs, providing useful performance visualization analyses and displays. Like many performance analysis and auto-tuning tools, TAU relies on PAPI for retrieving performance counter measurements. TAU also offers the functionality to profile so-called *user-defined events*. The meaning of these events is entirely determined by the user. Unlike PAPI's SDE effort, however, TAU's user-defined events are limited to single-value events and are specific to TAU only.

Another related project is Caliper [25], which offers a source-code annotation API for program instrumentation and performance measurement. Caliper is primarily aimed as a tool for performance experts to bake performance analysis capabilities directly into the applications they are trying to study. Among other performance values, such as timers, Caliper reads PAPI counters, so it can work synergistically with PAPI SDE by enabling performance experts to query library specific SDEs through Caliper.

VI. Conclusions

PAPI has provided a unification layer for hardware-based events, and enabled application developers and performance toolkits to access these events in a uniform and consistent way for more than 15 years. This paper presents our latest SDE developments that allow PAPI to perform the same role for software-based events. The addition of SDE in PAPI enables developers of libraries, application components, and runtime systems to expose internal, performance-critical information about their software in a consistent and standardized way.

The SDE integrations which we discussed in this paper highlight the importance of the different types of SDEs and their versatility for a wide variety of software layers such as PaRSEC and HPCG. The overhead analysis demonstrated that even for the most expensive SDE functionality (*Recorder*) the monitoring overhead is very low (tens of nanoseconds) under extreme use with benchmarks, and inconsequential in realistic usage scenarios.

In summary, scientific application developers can monitor SDEs together with traditional hardware performance counter data to acquire a more complete picture of the entire application performance. Using PAPI SDEs, both types of events can be monitored without the need for users to modify their applications or learn a new set of library and instrumentation primitives.

References

[1] T. Islam, K. Mohror, and M. Schulz, "Exploring the MPI tool information interface: features and capabilities," *The International Journal of High Performance Computing Applications*, vol. 30, no. 2, pp. 212–222, 2016.

[2] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT: An OpenMP tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators. IWOMP 2013*, M. M. Rendell A.P., Chapman B.M., Ed. Springer, Berlin, Heidelberg, 2013, lecture Notes in Computer Science, vol 8122.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[4] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. J. Dongarra, "PTG: an abstraction for unhindered parallelism," in *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '14, New Orleans, Louisiana, USA, November 16-21, 2014*, 2014, pp. 21–30.

[5] R. Hoque, T. Herault, G. Bosilca, and J. J. Dongarra, "Dynamic task discovery in parsec: a data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA@SC 2017, Denver, CO, USA, November 13, 2017*, 2017, pp. 6:1–6:8.

[6] R. L. Graham, B. Barrett, G. M. Shipman, T. S. Woodall, and G. Bosilca, "Open MPI: a high performance, flexible implementation of MPI point-to-point communications," *Parallel Processing Letters*, vol. 17, no. 1, pp. 79–88, 2007.

[7] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[8] M. A. Heroux, J. Dongarra, and P. Luszczek, "HPCG technical specification," Sandia Nationalal Laboratories, Tech. Rep. SAND2013-8752, 2013.

[9] J. Dongarra, M. A. Heroux, and P. Luszczek, "A new metric for ranking high performance computing systems," *National Science Review*, 2016.

[10] Jack Dongarra and Michael A. Heroux and Piotr Luszczek, "The High-Performance Conjugate Gradients Benchmark," *SIAM News*, vol. 51, no. 1, pp. 12–12, January/February 2018.

[11] U. Trottenberg, C. W. Oosterlee, and A. Schüller, *Multigrid*, ser. (with guest contributions by A. Brandt, P. Oswald, and K. Stüben). London NW1 7BY, UK: Academic Press, A Harcourt Science and Technology Company, 2001.

[12] I. Yamazaki, M. Hoemmen, P. Luszczek, and J. Dongarra, "Improving performance of GMRES by reducing communication and pipelining global collectives," in *Proceedings of 31st IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2017)*, ser. 18th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2017) PDSEC-17, Buena Vista Palace Hotel, Orlando, Florida, USA, June 2, 2017, best paper award.

[13] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage," *ACM Transactions on Mathematical Software*, vol. 5, pp. 308–323, 1979.

[14] J. J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling, "Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, pp. 1–17, March 1990.

[15] Jack J. Dongarra and J. Du Croz and Iain S. Duff and Sven Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, pp. 18–28, March 1990.

[16] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 14, pp. 1–17, March 1988.

[17] Jack J. Dongarra and J. Du Croz and Sven Hammarling and R. Hanson, "Algorithm 656: An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 14, pp. 18–32, March 1988.

[18] J. J. Dongara, J. R. B. Cleve B. Moler, and G. W. Stewart, *LINPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1979.

[19] Ed Anderson and Z. Bai and C. Bischof and Susan L. Blackford and James W. Demmel and Jack J. Dongarra and J. Du Croz and A. Greenbaum and Sven Hammarling and A. McKenney and Danny C. Sorensen, *LAPACK User's Guide*, 3rd ed. Philadelphia: Society for Industrial and Applied Mathematics, 1999.

[20] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*, 2011, pp. 1432–1441.

[21] MPI Forum, "MPI: A Message-Passing Interface Standard Version 3.1," http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, June 4 2015.

[22] OpenMP Architecture Review Board, "OpenMP Application Program Interface, Version 4.0," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[23] OpenMP Tools Working Group, "OpenMP Technical Report 2 on the OMPT Interface," http://openmp.org/mp-documents/ompt-tr2.pdf.

[24] S. S. Shende and A. D. Malony, "The TAU Parallel Performance System," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006.

[25] Lawrence Livermore National Laboratory, "Caliper: Application Introspection System," https://computation.llnl.gov/projects/caliper.