Capturing Inter-Application Interference on Clusters

Aamer Shah and Felix Wolf
German Research School for
Simulation Sciences
Laboratory for Parallel Programming
{a.shah, f.wolf}@grs-sim.de

Sergey Zhumatiy and Vladimir Voevodin
M. V. Lomonosov
Moscow State University
Research Computing Center
{serg, voevodin}@parallel.ru

***Abstract*—Cluster systems usually run several applications—often from different users—concurrently, with individual applications competing for access to shared resources such as the file system or the network. Low application performance is therefore not always the result of inefficient program design, but may instead be caused by interference from outside. However, knowing the difference is essential for an appropriate response. Unfortunately, traditional performance-analysis techniques consider an application always in isolation, without the ability to compare its performance to the overall performance conditions on the system when it was executed. In this paper, we present a novel approach of how to correlate the performance behavior of applications running side by side. To accomplish this, we divide the application runtime into fine-grained time slices whose boundaries are synchronized across the entire system. Mapping performance data related to shared resources onto these time slices, we are able to establish the simultaneity of their usage across jobs, which can be indicative of inter-application interference. Our experiments show that such interference effects, for which the developer is usually not to blame, can degrade application performance significantly.**

I. Introduction

Many cluster systems are operated in space-sharing mode. Unless an application requires all cluster nodes for itself, the system is divided among several concurrently running jobs, with the exact partitioning determined by the job scheduler. Depending on the precise configuration, an application usually owns only some of the resources such as the CPUs it is running on exclusively, while others such as the file system and the network are shared among one or more jobs. Naturally, the performance behavior of a job using such shared resources depends not only on its own usage pattern but also on the load that other shareholders impose at the same time. For example, two jobs that write large data sets concurrently may see less file-system bandwidth than two that serialize their I/O. This phenomenon can be thought of as application jitter. While jitter coming from the operating system [1]–[4] received much attention in the past, inter-application interference on clusters is known but has so far not been subject to systematic investigation. However, its asynchronous nature can hamper scalability in a similar way.

In spite of this situation, traditional performance-analysis tools [5]–[8] still consider an application from a first-person perspective. They collect performance data exclusively for one application with the implicit assumption that it runs in isolation. The only way to account for interference with such tools is to run the application multiple times and to quantify run-to-run variation. Otherwise, foreign load on shared resources cannot be blamed for reduced performance. And even then the precise nature of this interference remains opaque. Although approaches [9], [10] exist to silently create performance profiles of all jobs running on a system and thus to capture all those that share the system at a given time, the ability to establish the simultaneity of performance events, a prerequisite for the correlation of performance events across jobs, is still missing.

In this paper, we present a new workload-oriented performance-analysis approach that can be used to detect and present signs of application interference. Like in earlier scenarios, we capture basic performance data from every job of the system. The novelty, however, is to divide the time on the system into small slices for which we record the performance data separately. The slices are thin enough so that even smaller performance events such as sudden peaks of file I/O in one application can be correlated with temporary performance degradation observed for another. Different from incremental profiles of individual jobs [11], the slice boundaries are synchronized across the whole system to ensure that the slices of different jobs running at the same time can be precisely mapped onto each other.

The remainder of the paper is organized as follows: In Section II, we explain our approach in more detail and introduce LWM^2, a monitoring module that collects time-sliced performance measurements from applications, sketching its design and deployment. The experimental evaluation in Section III quantifies LWM^2's runtime overhead before analyzing interference effects at the level of the file system, of the network, and of shared nodes. Finally, we draw conclusions and define future research directions in Section IV.

II. Architecture

To capture application interference, we model the operation of a cluster system as a time-space grid, as illustrated in Figure 1. The space dimension represents the hardware, which is divided into a set of disjoint nodes. Each node may run one or more processes. The time dimension represents the runtime of the system, which is divided into a set of disjoint time slices. Discrete time slices are needed to establish an approximate notion of simultaneity among performance phenomena. Assuming static allocation of hardware resources, each job on the system occupies one or more rectangular areas that all start at the same time and end at the same time. Performance data is collected at the granularity of processes and time slices. Thus, for each time slice and process, we record a vector of performance metrics, resulting in as many metric vectors per node as there are processes on the node during the time slice.

978-1-4799-0898-1/13/$31.00 ©2013 IEEE

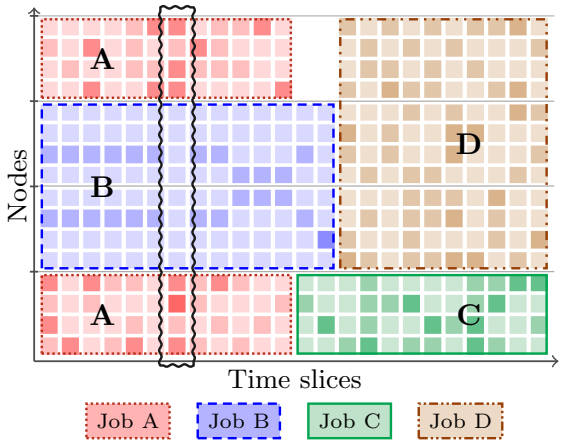


Fig. 1. Operation of a cluster system as a time-space grid. The picture shows four nodes, each running four processes (denoted by colored squares) at a time. The brightness of each square indicates the intensity of a performance metric. In the highlighted time slice, high intensity in job A coincides with low intensity in job B.

Now we can correlate performance data in two different ways: First, we can check whether we can see differences between different nodes that remain stable across an extended period of time. Although outside the scope of this paper, this information can be used to detect hardware anomalies that do not constitute outright failure but lead to some form of performance degradation applications cannot be blamed for. Examples include network links whose throughput is reduced due to cabling problems, a node throttled due to some error condition, or a smaller amount of per-node memory. Second, and this is central to our approach, we can compare the performance of an application during a given time slice with the performance background, that is, the performance of applications running at the same time on relevant parts of the system. Since the possibility of interference depends on the topological characteristics of the system, we also precisely record the node each process is running on.

We implement the proposed approach by collecting performance data from every job running on the system. For this purpose, we designed a lightweight monitoring module called *LWM*^{2} that is supposed to be dynamically linked to each application at job start—with the option of being disabled if needed. Our monitoring module collects basic performance metrics for each process, without modification of the executable and with low time and space overhead. We achieve $\leq 1\%$ runtime penalty and ≤ 5 MB of required buffer space per process and day.

The monitoring module features a modular architecture for customization and further extendibility, offering compile time customization of profiling capabilities with support for further configuration of the compiled capability through environment variables. Currently developed capabilities include profiling of MPI applications, Pthreads-based multithreaded applications, and CUDA applications through CUPTI [12], along with the interception of POSIX file I/O calls. Support for Xeon Phi accelerators is in progress. Finally, sequential performance is measured through hardware counters using PAPI [13].

1) Instrumentation: To keep runtime dilation low, LWM^2 employs direct instrumentation via interposition wrappers but without making any time measurements inside, as their cost can accumulate in application that frequently call any of the wrapped functions. Interposition wrappers enclose all calls to MPI and POSIX file I/O functions while CUPTI's callback functions are used for CUDA calls. The direct instrumentation is used to count function calls and to intercept data-transfer parameters such as the number and size of messages, the data volumes read from or written to disk, or the amount of data transferred between hosts and accelerators. Hardware counters are recorded when crossing time-slice boundaries. In addition, we apply sampling to estimate the time spent in certain classes of functions. For this purpose, we earmark the wrapped routine currently being executed so that we can correctly attribute samples to them without having to examine the stack.

2) Time slices: In addition to creating a job digest at the end that summarizes the whole execution in terms of performance, LWM^2 creates separate profiles for each time slice, which allow insights into the changing performance dynamics of the application. As illustrated in Figure 1, the time slices are synchronized across the system using system time so that it is possible to compare and aggregate performance data both across all processes of a parallel job and across all jobs running on the system at a given time. In essence, time slices offer a way to establish the simultaneity of performance incidents such as file-system access storms. While we left the length of the slices configurable, we chose a length of 4 s for the purpose of this study. This is small enough to capture performance dynamics with reasonable granularity but still large enough in relation to the precision at which the system time is usually synchronized (in the order of 1 ms without global clock). With performance data collected at the granularity of time slices, a user can now compare the performance dynamics of the own application with the performance background and identify correlations.

3) Multithreaded applications: Collecting time-sliced performance data for multithreaded applications is not trivial. Specific challenges arise when trying to reconcile small memory footprint with thread safety and low runtime dilation. We take a number of steps to ensure a light and portable solution. First, we manage threads at Pthreads level, which is the foundation of many higher-level threading libraries. Second, we aggregate the performance metrics of individual threads *in situ* at the end of each time slice. We use a dual-buffer storage system for threads, one buffer to collect live performance data and second to allow data aggregation at time slice boundaries. The roles of the buffers switch every time slice.

However, aggregating thread data at time slice boundaries is difficult. Signaling the end of a time slice via a timer interrupt is not an option as using mutexes in an interrupt is prohibited. We employ an auxiliary *heartbeat thread* to solve this dilemma. The heartbeat thread is created during LWM^2's initialization and is activated only at time slice boundaries. When activated, the heartbeat thread first creates new buffer space for the new time slice. Then, it switches the active buffer in the dual-buffer storage system for thread data. As a last step, it aggregates performance metrics of individual threads into LWM^2's linearly increasing buffer space. The *in situ* aggregation means LWM^2 need less than 5 MB space per process and day. The inability to use mutexes in a signal handler is also the reason why we slice only the metrics

collected in wrappers, which however are sufficient to capture the use of shared resources. The timings gained through sampling are left to the execution digest, where they provide a classic performance summary for the entire execution.

III. Evaluation

Our evaluation pursues several objectives: First, we confirm LWM^2's low runtime overhead. Second, we demonstrate how it can be used to identify application interference. At the same time, we analyze the interference potential of typical shared resources.

All our tests were performed on two machines of the Jülich Supercomputing Centre: JUROPA is a Linux cluster consisting of 2208 compute nodes, each equipped with two Intel Xeon X5570 (Nehalem-EP) quad-core processors running at 2.93 GHz. The nodes are connected through an Infiniband QDR network with non-blocking fat-tree topology. All our file I/O was performed on a Lustre file system with four meta-data servers (MDS) of type Bull NovaScale R423-E2 (two Nehalem-EP quad-core & two Westmere-EP, 6-core) and eight object storage (OST) servers of type Bull NovaScale R423-E2 (Westmere-EP, 6-core) attached to the same Infiniband network. JUDGE is a Linux GPU cluster. The node we used in this study features two Intel Xeon X5650 (Westmere) 6-core processor running at 2.66 GHz plus two Nvidia Tesla M2050 Fermi GPUs.

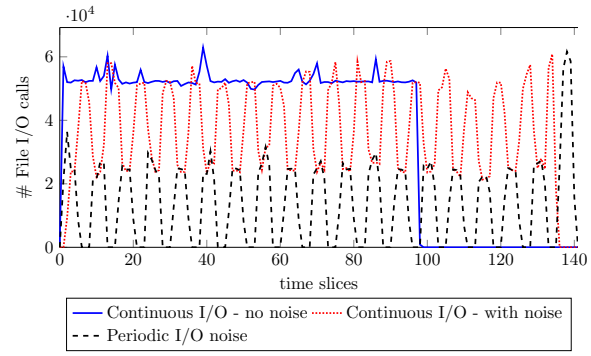
A. Overhead

Since silent profilers such as LMW^2 are supposed to be operated in a compulsory mode, requiring at least an explicit opt-out to be disabled, low overhead is essential for user acceptance. The overhead of a profiler falls in two categories—space and time. We collect currently 22 metric values á 8 bytes per time slice (i.e., every 4 s) and process, resulting in a memory footprint of less than 4 MB for 24 hours of execution. The runtime dilation was verified using applications from two benchmark suites: SPEC MPI 2007 [14] for single-threaded MPI programs and the NAS Parallel Benchmark [15] for hybrid MPI/OpenMP programs. The profiling overhead was measured for alternated profiled and non-profiled execution and repeated 30 times. After removing outliers with the modified Z-score method [16], the mean overhead for almost all cases was less than 1%. An anomalously high overhead behavior was exhibited only by 143.dleslie of the SPEC MPI 2007 benchmark suite. Further investigation identified the PAPI library as the source of this deviation. A run without PAPI library resulted in the expected mean overhead of less than 1%.

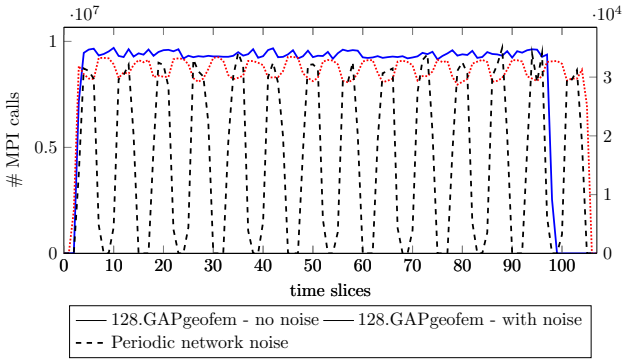
B. Interference

In spite of running in space-sharing mode, applications on cluster systems usually share some resources that they cannot own exclusively. Contention for such shared resources may cause inter-application interference, one component of the overall runtime jitter. In a set of experiments, we measured the jitter resulting from this interference for different types of shared resources.

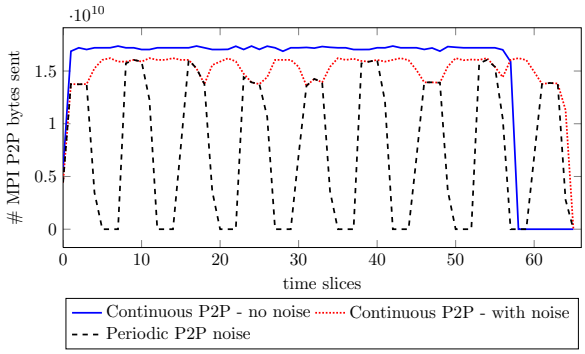
Concurrently executing applications usually share at least the file system and depending on its topology parts of the network. Unless the file system is reached over a dedicated



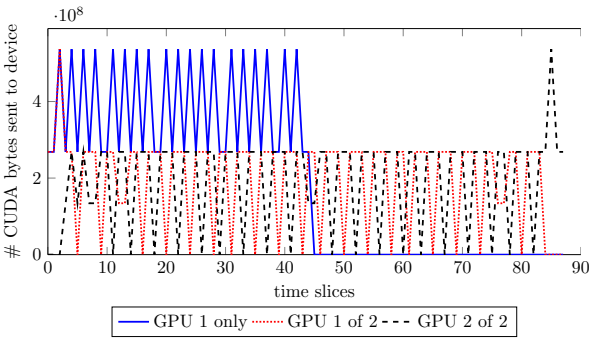
(a) Continuos I/O benchmark vs. periodic I/O noise.



(b) 128.GAPgeofem vs. periodic network noise.



(c) Communication interference when sharing nodes.



(d) Two GPU benchmarks sharing one PCI Express bus.

Fig. 2. Inter-application interferences when sharing a variety of resources.

network, file I/O also implies network traffic that may interfere with network traffic between compute nodes. While some systems allocate nodes exclusively to a single job, others such as Tsubame 2 at Tokyo Tech may even allow the sharing of nodes. There, node sharing is motivated by the desire to balance the need for classic CPUs and GPUs among a highly diverse workload. If nodes are shared then contention may occur with respect to the network interface, the memory, and, depending on the architecture, the accelerator bus. We assume a sensible node-sharing scheme in which applications run at least on separate sockets with their physically attached memory, hence already minimizing memory-subsystem interference. We target each of these shared resources in experiments to measure their impact as a source of interference.

In production environments, cluster applications experience jitter from many sources. Extracting and measuring the contribution of contention for a single resource to inter-application interference is therefore challenging. To do exactly this, we created a set of benchmarks, one for each resource type, that inject noise according to a distinct periodic pattern of resource usage. If the benchmark interferes with another simultaneously executing application that makes use of the same resource, it will impose the inverted pattern on the time-sliced profiles of the application. That is, whenever the noise exhibits a peak, the metrics of the application suffer a dent. Note that because each metric value represents a rate (per time slice), the slowdown affects all metrics. Thus, we identify interference through inspection of time-sliced profiles. Moreover, we quantify the interference effect by comparing the execution time under noise with the time we see without introducing noise. Since LWM^2 is still under test and at the time of writing not mandatory for all jobs on the system, we could not profile the complete workload—just the jobs we submitted ourselves. However, the artificial noise pattern is distinct enough to make coincidence extremely unlikely.

1) I/O subsystem: On JUROPA, file I/O operations may share the communication network and I/O nodes. At the same time, the network is also available to messages exchanged between compute nodes. To study the interference potential of file I/O operations, we ran two I/O intensive benchmarks side-by-side, one with a continuous I/O pattern, and one with a periodic I/O pattern. We ran each of the benchmarks with 256 processes on disjoint sets of 32 nodes of JUROPA. In the first benchmark, which we call the probe, each process continuously opened a file, wrote a 100×100 integer matrix in consecutive write operations to it, one per element, and closed it again. Each process used its own file. It was created in the beginning and then re-used in each iteration. The second benchmark, the noise, did the same, only that the stream of matrix writes was periodically interrupted by phases of inactivity. The benchmarks performed predominantly file I/O. Hence, any resulting interference pattern can be attributed to this I/O with high confidence. Figure 2(a) shows the number of write operations per time slice, accumulated across all processes, as an indicator of performance. The imprint the noise left on the probe as a sign of interference is clearly discernible, especially when compared to the execution without noise. The interference reduced the application's write performance significantly, with a drop of 50% at the points of interference and an overall execution time prolonged by 35%. The experiment was repeated several times and the same behavior was observed each time. Exposing 128.GAPgeofem from the SPEC MPI2007 suite, a finite-element code that performs I/O on a regular basis, to our artificial noise resulted every time in an abort with a "file not accessible" error, indicating that I/O subsystem interference can even be prohibitive in certain cases.

2) Communication network: Although we experimented with a large variety of communication patterns and execution configurations, we were surprisingly unable to find any unequivocal evidence of significant interference at the level of message exchange between JUROPA's compute nodes. This might have to do with the generous bandwidth and the well-balanced fat-tree topology of JUROPA's network and does therefore not necessarily apply to other systems. On the other hand, advanced interconnect designs such as Cray's Aries suggest that pure network interference between applications will play a minor role in the future anyway.

Nevertheless, what we found is a reproducible case of interference between 128.GAPgeofem, which performs both inter-process communication and file I/O in regular intervals, and a noise benchmark that alternates between periods of intensive global inter-process communication and periods of silence. Each program was executed on a separate set of 128 nodes of JUROPA. This system has a two-layer fat-tree topology, where each leaf switch connects 24 compute nodes and the top level switches connect all the leaf switches. Utilizing 128 nodes guaranteed that the two programs used overlapping portions of the network. Figure 2(b) shows the number of collective calls per time slice with and without noise. It can be seen that during the network activity phase of the periodic noise benchmark, the performance of 128.GAPgeofem drops by 10%. This leads to an overall 5% longer execution time compared to a run without noise. The experiment was repeated several times and each time 7% to 10% degradation was observed during the active phase of the periodic noise benchmark. While this indicates that the interference occurs at the level of the network, it remains unclear whether the source of the interference was inter-process communication or file I/O. During the entire execution, each process of 128.GAPgeofem exchanges more than 1 GB of data in MPI point-to-point communication (and a smaller amount in collectives), while it writes only 0.5 MB of data to files. On the other hand, when exposing 128.GAPgeofem to network noise, we observed several failures due to the inaccessibility of the file system. Also in the light of our disability to demonstrate pure network interference, we are therefore inclined to attribute this phenomenon to cross-interference between network communication and file IO.

3) Node: To improve resource utilization, some accelerator-based HPC systems allow individual nodes to be shared between CPU- and GPU-oriented applications. An example is Tsubame 2 at Tokyo Institute of Technology, which permits node sharing on its thin-nodes, consisting of two host processors and three GPUs.

a) Network interface: Even if simultaneously executing applications do not share individual sockets, a common scheduling constraint in the interest of minimizing interference within the memory subsystem, the applications share at least the network interface of the node. In our experiments, we tried

to investigate whether such sharing can lead to significant interference. Since none of our test systems allowed nodes to be shared between applications, we split the world communicator of a single benchmark to mimic two different applications. The communicator was split in such a way that the border between the two communicators divided nodes but assigned the sockets exclusively either to one or the other. Both communicators performed MPI point-to-point communication, the first one (i.e., the probe) in a continuous fashion, the second one (i.e., the noise) only periodically, that is, interrupted by phases of inactivity. Figure 2(c) shows the number of bytes sent per time slice for each communicator. For comparison, the probe communicator was also measured with the noise communicator disabled. We observed sporadic interference between the two communicators during execution. We ran the experiment several times and observed sporadic interference in each of the runs, albeit occurring at different points. On average, a run with ten bursts of noise lead to interference during four or five of these bursts. Thus, the average probability of interference for a single burst was between 0.4 and 0.5. A run of the benchmark with only collective calls did not produce any interference.

b) PCI Express bus: Modern heterogeneous clusters may feature more than one accelerator per node. If they are attached to the same PCI Express bus like on JUDGE then interference can occur when two GPU applications share a node even if they do not share GPUs. This is also true for a single application with multiple threads that use different GPUs independently. To test this hypothesis, we ran a simple benchmark on JUDGE, repeatedly multiplying matrices on a single GPU. For each multiplication, the benchmark, which was sequential on the host side, copied the matrix from the host memory to the device. Two instances of the benchmark were executed together on a shared node and compared with a single-instance run. Figure 2(d) shows the number of bytes transferred from host to device per time slice as an indicator of overall progress. It can be seen that for shared execution, the performance of the application drops by up to 50%, resulting in 35% longer execution. To eliminate the host memory as the source of interference, we replaced the second GPU benchmark instance with another benchmark that accessed the host memory in a way similar to the GPU benchmark but without accessing the PCI bus. No performance degradation was observed in this case. This strongly points to the shared data bus as the bottleneck and source of performance degradation.

IV. Conclusion and Future Work

To analyze inter-application interference on clusters, we presented a novel profiling methodology based on the concept of globally synchronized time slices that allows the correlation of performance phenomena across jobs. By observing probe applications exposed to a pronounced periodic noise pattern, we analyzed the interference potential of several shared resources. We found file I/O to be a major catalyst of interference with I/O performance degraded by up to 50%. We further identified slower file I/O as a likely effect of concurrent inter-process communication. Finally, we found evidence of bottlenecks at the host channel adapter and the PCI Express bus that connects host and device on heterogeneous clusters when sharing nodes between applications. The latter can be easily addressed by investing in a separate bus.

With file I/O having crystallized as a central theme of inter-application interference, we want to study especially I/O interference patterns at greater detail, modeling its cost in terms of time and energy, and looking for ways to avoid it, for example, by designing I/O-aware scheduling policies.

Acknowledgement: This work has been supported by the European Commission under Grant No. FP7-277463 and by the Russian Ministry of Education and Science under Grant No. 07.514.12.4001.

References

[1] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proc. of the ACM/IEEE Supercomputing Conference, Phoenix, AR, USA*, November 2003.

[2] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proc. of the ACM/IEEE Supercomputing Conference, New Orleans, LA, USA*. IEEE Computer Society, November 2010.

[3] S. Seelam, L. Fong, J. Lewars, J. Divirgilio, B. Veale, and K. Gildea, "Characterization of system services and their performance impact in multi-core nodes," in *Procs. of the IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2011, pp. 104–117.

[4] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proc. of the ACM/IEEE Supercomputing Conference, Austin, TX, USA*. IEEE Computer Society, November 2008.

[5] 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, April 2010.

[6] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1.

[7] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A tool to visualize and analyze parallel code," in *Procs. of WoTUG-18: Transputer and Occam Developments*, April 1995, vol. 44, pp. 17–31.

[8] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[9] K. Fürlinger, N. Wright, and D. Skinner, "Effective performance measurement at petascale using IPM," in *IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS)*, December 2010, pp. 373 –380.

[10] R. Kufrin, "PerfSuite: An accessible, open source performance analysis environment for Linux," in *6th International Conference on Linux Clusters: The HPC Revolution, Chapel Hill, NC, USA*, 2005.

[11] K. Fürlinger, M. Gerndt, and J. Dongarra, "On using incremental profiling for the performance analysis of shared memory parallel applications," in *Euro-Par 2007 Parallel Processing*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4641, pp. 62–71.

[12] "CUDA toolkit documentation: CUPTI http://docs.nvidia.com/cuda/cupti."

[13] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Procs. of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[14] "Standard performance evaluation corporation. (2007) SPEC MPI2007 benchmark suite. http://www.spec.org/mpi2007/."

[15] R. F. van der Wijngaart and H. Jin, "The NAS parallel benchmarks, multi-zone versions," no. NAS-03-010, June 2003.

[16] B. Iglewicz and D. Hoaglin, *How to detect and handle outliers*. ASQC Quality Press (Milwaukee, Wis.), 1993, vol. 16.