Procedia Computer Science

Volume 80, 2016, Pages 1507–1518

ICCS 2016. The International Conference on Computational Science

WOWMON: A Machine Learning-based Profiler for Self-adaptive Instrumentation of Scientific Workflows

Xuechen Zhang[1], Hasan Abbasi[2], Kevin Huck[3], and Allen D. Malony[3]

[1] Washington State University Vancouver, Vancouver, Washington, U.S.A
xuechen.zhang@wsu.edu
[2] Oak Ridge National Laboratory, Oak Ridge, Tennessee, U.S.A.
habbasi@ornl.gov
[3] University of Oregon, Eugene, Oregon, U.S.A
khuck@cs.uoregon.edu, malony@cs.uoregon.edu

Abstract

Performance debugging using program profiling and tracing for scientific workflows can be extremely difficult for two reasons. 1) Existing performance tools lack the ability to automatically produce global performance data based on local information from coupled scientific applications of workflows, particularly at runtime. 2) Profiling/tracing with static instrumentation may incur high overhead and significantly slow down science-critical tasks. To gain more insights on workflows we introduce a lightweight workflow monitoring infrastructure, WOWMON (**WO**rkflo**W** **MON**itor), which enables user's access not only to cross-application performance data such as end-to-end latency and execution time of individual workflow components at runtime, but also to customized performance events. To reduce profiling overhead, WOWMON uses adaptive selection of performance metrics based on machine learning algorithms to guide profilers collecting only metrics that have most impact on performance of workflows. Through the study of real scientific workflows (e.g., LAMMPS) with the help of WOWMON, we found that the performance of the workflows can be significantly affected by both software and hardware factors, such as the policy of process mapping and in-situ buffer size. Moreover, we experimentally show that WOWMON can reduce data movement for profiling by up to 54% without missing the key metrics for performance debugging.

Keywords: TAU, Scientific workflows, Performance monitoring

1 Introduction

Data-intensive knowledge discovery requires scientific applications to run concurrently with analytics and visualization codes as *workflows*. These coupled applications can be executed in situ for timely output inspection and knowledge extraction. Performance debugging for scientific workflows can be more difficult than that for stand-alone applications using existing

doi:10.1016/j.procs.2016.05.474

high-performance computing (HPC) performance tools [20, 2] for two reasons. First, they lack the ability to produce global performance data based on local data from multiple coupled applications at runtime. Without the global data, for example, it is difficult to observe the total amount of time that a scientific workflow used to process data produced by an application at a particular execution step. We refer to this time as *end-to-end* latency of a workflow. Second, profiling/tracing of multiple applications with static instrumentation can incur high storage overhead and slow down the execution of mission-critical scientific tasks.

In this paper we describe a novel monitoring infrastructure called WOWMON to gain deeper understanding of *where* and *when* performance bottleneck happen in scientific workflows. It provides three unique features: 1) *online feedback* for timely workflow tuning and adaptation by accessing in-memory performance data; 2) *global monitoring* which produces global performance data, specifically used for performance debugging of workflows; 3) *self-adaptive profiling* which can guide profilers collecting only the metrics that have most impact on performance of workflows using machine learning algorithms. More specifically, we made the following contributions in this work:

- We designed simple interface for users to access in-memory performance data across multiple applications/components of workflows, and implemented a flexible and lightweight runtime system, supporting data sampling, multiple network topologies, metric selection for customized coverage of performance metrics, and self-adaptive profiling.

- We implemented WOWMON based on TAU [20] and EVpath [4] libraries. With the help of WOWMON our experiments using realistic scientific workflows (e.g., *LAMMPS*) show that the end-to-end latency of workflows can be affected by the memory buffer size for in-situ operations and the policy of mapping processes of applications to nodes.

- We demonstrated that WOWMON can evaluate performance metrics at runtime using feature selection algorithms, and then use its output to further reduce the volume of performance data by 54% for the *LAMMPS* workflow, while still providing in-depth insights for end-users.

The rest of this paper is organized as follows. Section 2 discusses the existing tools related to performance research of scientific applications. Section 3 describes the design and implementation of WOWMON. Section 4 describes and analyzes experimental results. Finally, we conclude the paper in Section 5.

2 Related Work

Most of the previous work focused on the performance measurement and monitoring of standalone applications. For example, TAU [20] has been very popular as an HPC toolkit for performance instrumentation, measurement, analysis, storage, and visualization. By leveraging PAPI [2] and source instrumentation, it can access both hardware counters and application performance data in standard ways. However, TAU's performance data analysis is typically performed after data has been written to disk-resident logs, or in a central location. This restricts the scalability of online monitoring and analysis tasks.

An online monitoring infrastructure is needed to capture transient effects of data-driven behaviors. Although many works have been proposed for this purpose, none of them was designed for capturing performance of workflows. With respect to instrumentation, a salient feature of the performance measurement tool Paradyn [13] is flexible selection of metric subsets for binary instrumentation at runtime, which might take seconds to finish. In contrast, WOWMON creates initial instrumentation points based on a priori knowledge using the simple WOWMON API. WOWMON's runtime library supports online reconstruction of metric sets in networking layers for reducing monitoring overhead. With respect to online data access, TAUg [10] was designed to collect performance data of MPI processes for stand-alone applications. Roth et al. [19] presented an online performance diagnosis system, focusing on optimization of problem search strategies for finding bottlenecks of large-scale stand-alone applications. The Ganglia distributed monitoring system [12] is widely used in both enterprise and HPC domains. However, it only targets on system metrics such as CPU idle time. WOWMON needs to collect source-level data (e.g., function execution time) as well as track system-level events because both are critical for the analysis of the workflow performance as shown in Section 4.

Other application-steering software may have built-in monitoring modules. An example is Peridot [6], which allows programmers to guide analysis process with query and then interacts with runtime systems. WOWMON can also provide feedback for optimization of the metric sets. Moreover, it provides key services for performance diagnosis of a workflow not a stand-alone application. For the latter, performance steering has been well studied. Autopilot [18] used classification algorithms to produce the hints of making resource management decision based on collected performance data. WOWMON can also leverage learning algorithms [9] in order to evaluate importance of metrics. Eisenhauer et al. [3] proposed a steering system working with multiple collaborated users. Falcon [7] supports online interaction with end users of complex scientific simulations. Tapus et al. [21] proposed a steering approach, which takes into account the performance characteristics of linked libraries of binaries.

In summary, WOWMON is the only system, which is designed for performance diagnosis and steering of the scientific workflows composed of in-situ analytics.

3 The Design of WOWMON

We have three objectives in the design of WOWMON: 1) accessing in-memory data structures for collecting online performance data; 2) tracking global performance data using the local data collected from individual workflow components; 3) providing a flexible and lightweight networking layer which can support self-adaptive profiling/tracing using machine learning algorithms. In addition, WOWMON provides simple APIs for end users to create and track timers, specify system and application events of interests, and customize communication patterns. We will discuss them in detail in the following sections.

3.1 Architecture Overview

The WOWMON infrastructure includes three major components, user interface (APIs), a runtime library, and a workflow manager. Programmers need to instrument source code using the APIs, as discussed in Section 3.2. At runtime, WOWMON creates relay networks connecting the processes of applications and the workflow manager using a network topology specified in the parameter ($WOWMON\_TOPO\_COMM$) by users (*star* topology is set by default). Then software and hardware events can be added to an initial metric set ($MetricSet_{init}$), whose members might be dynamically selected at runtime to control the size of the set, which correlates to monitoring overhead. Specifically, the software events can track program's performance such as function execution time, call depth, and memory heap size when entering and exiting a function. And the hardware events are used to track hardware performance such as cache

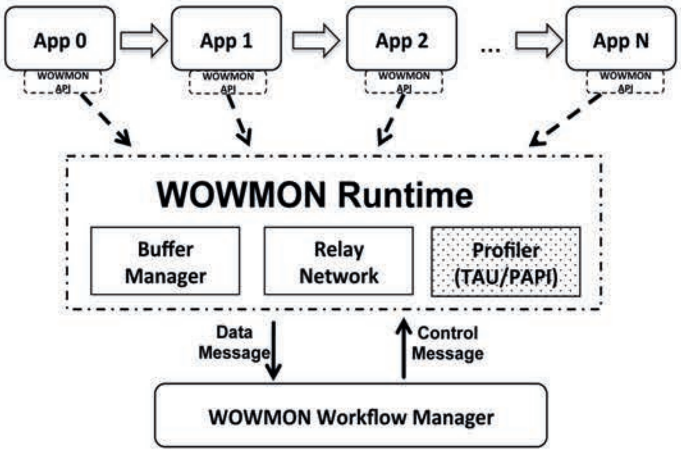

Figure 1: WOWMON architecture

misses, floating point operations, and so on. WOWMON reads the values of the metrics from a in-memory data structure of a profiler, such as TAU [20]. The data structure tracks all the current value of the metrics in $MetricSet_{init}$. For those not strongly correlated to the end-to-end latency of workflows, as indicated by the outputs of machine learning algorithms, the workflow manager can at runtime specify another set of metrics $MetricSet_{interest}$ to replace $MetricSet_{init}$. $MetricSet_{interest}$ needs to be a subset of $MetricSet_{init}$. And only performance data for the metrics in $MetricSet_{interest}$ are transferred to the workflow manager. Because the size of $MetricSet_{interest}$ may be significantly smaller than $MetricSet_{init}$, the monitoring overhead of WOWMON can be reduced as shown in Section 4.4.

Function	Description
WOWMON_REGISTER_VIEW()	Establish connection to runtime
WOWMON_ADD_EVENTS()	Track hardware/software events
WOWMON_GET_GLOBAL_DATA()	Notify the networking layer to send data
WOWMON_DEREGISTER_VIEW()	Disconnect from the runtime
WOWMON_INIT_TIMER()	Create a user timer
WOWMON_TRACK_TIMER()	Track a user timer

Table 1: A description of WOWMON APIs

3.2 User Interface

The user interface (APIs) of WOWMON is designed to be easy to use, and callable from C, C++, and FORTRAN. The functions listed in Table 1 work synergistically with WOWMON runtime to fulfill user's needs on performance monitoring. The APIs include six functions. WOWMON_REGISTER_VIEW() creates data buffers and connections to the runtime system. With this function users can specify a network topology and name a communicator. WOWMON_ADD_EVENTS() adds events to $MetricSet_{init}$. For example, a software event can be added to track memory heap size when entering a particular function having the signature string "void compute_and_send(bond_record_ptr) [bonds.c 398,1-469,1]" [1]. Function signatures can be obtained using the existing profiling tools, such as TAU. WOWMON_GET_GLOBAL_DATA() is called to send performance data to the workflow manager. For data sampling users can specify a subset of processes as the samples of profiling data using WOWMON_PATTERN, which should be in the range of 0 to *nprocs*, which is the total number of processes of an application. When it is set to i, only data from the process of rank 0 to $i-1$ are collected and sent to the workflow manager. By default it is set to *nprocs*. This feature is very useful for reducing communication overhead when workflows are executed at scale with thousands of processes. Data collection can be disabled by setting WOWMON_PATTERN to 0 when users are not interested in the metrics from a particular component or the latency between directly connected components. When a program is finalized, WOWMON_DEREGISTER_VIEW() is needed to disconnect from runtime and clean up state buffers. WOWMON_INIT_TIMER() and WOWMON_TRACK_TIMER() are called to create and track a user timer placed at a specific location in source code.

3.3 The Runtime Library

In the design of WOWMON runtime, we optimize its profiling operations, buffer management, and communication considering both portability and scalability. Three components, including a *profiler*, a *buffer manager*, and a *relay network*, interact with the workflow manager to achieve lightweight monitoring, as shown in Figure 1.

The *profiler* maintains in-memory data structures to keep tracking of the values of timers and events instrumented by programmers. WOWMON uses the existing profiling software such as TAU and PAPI for this purpose, although other tools can be supported easily because only ~50 lines of code are profiler-dependent. When TAU is used as the *profiler*, WOWMON calls two key functions TAU_GET_EVENT_NAMES() and TAU_GET_FUNC_VALS() [22] to read the values of the metrics in $MetricSet_{interest}$ and stores them in data buffers.

The *buffer manager* tracks $MetricSet_{interest}$ and $MetricSet_{init}$ for each application of a workflow. Based on $MetricSet_{interest}$ it reads in-memory profiles, creates buffers for data messages, and then sends those messages to the workflow manager. As a receiver, it also updates $MetricSet_{interest}$ when control messages for adaptive profiling are received. The format of the data messages is composed of both data fields, each corresponds to a metric in $MetricSet_{interest}$, and meta-data fields such as *application ID*, *process ID*, and *communicator ID*, which are then used by the workflow manager to identify the messages from various running instances and produce global performance data (e.g., end-to-end latency of workflows) based on profiled local information.

The *relay network* is a thin communication layer between the runtime of applications (sources) and workflow managers (sinks). The messages sent from the source to sink nodes may be processed through multi-level relay nodes in order to remove a networking bottleneck caused by a centralized workflow manager and improve the scalability of the network. EVpath [5] is selected for relay network management for its proven performance on high-end machines and support of multiple networking drivers (e.g., TCP/IP, Infiniband [14], and ENET [17] etc). If *WOWMON_TOPO_COMM* is set to *star*, all the sources are directly connected to the workflow manager. This topology should only be used for small scale runs. If *WOWMON_TOPO_COMM* is set to *tree*, a network having a *spanning tree* topology is created for the applications having more than 128 processes. In this scenario, only a limited number of relay nodes are directly

[1]compute_and_send() is an expensive function in the *Bonds* code of the *LAMMPS* workflow. More details of the workflow are presented in Section 4.1.

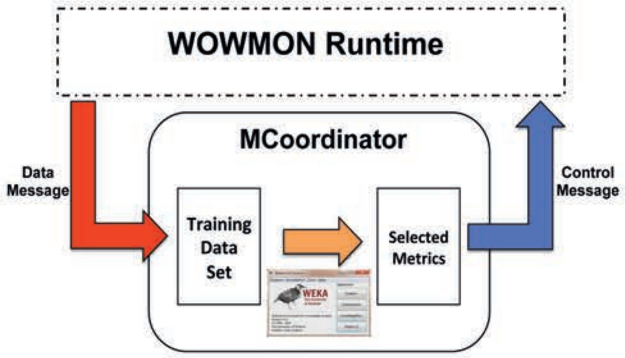

Figure 2: An example of a workflow manager using machine learning algorithms provided by Weka to select correlated metrics.

connected to the workflow manager. The relay nodes periodically send the summarized performance data received in a period to the workflow manager for reduced overhead of data transfer and a more balanced load distribution.

3.4 Self-adaptive Profiling

A workflow manager, named $MCoordinator$, is designed to help select metrics according to their impact on end-to-end latency, rather than acting only as a dummy manager just collecting local data. As shown in Figure 2, $MCoordinator$ collects data from workflow components and then converts them to software-compatible formats, such as attribute-relation file format (ARFF) required by the Weka [8] software. In the next step, attributes selection algorithms are executed to rank the metrics in $MetricSet_{init}$ or $MetricSet_{interest}$. Specifically, we use an algorithm called *CfsSubsetEval* in Weka to evaluate the worth of a subset of metrics by considering the individual predictive ability of each metric along with the degree of redundancy between them. Subsets of metrics that are highly correlated with the latency while having low inter-correlation are preferred [9]. Breadth-first search algorithm [11] is used for metric subsets creation during the search. In the end, the metrics in the selected subset are sent back to WOWMON runtime via a control channel. $MCoordinator$ can trigger the metric selection procedure both offline and online. In addition, it needs to decide when to trigger the procedure to minimize its learning overhead. A hybrid approach is used in the design of $MCoordinator$, whose metric selection procedure can be triggered by both timers and special events, such as a moving average of the end-to-end latency of workflows is 50% higher/lower than the one measured in the previous steps.

4 Evaluation

In this section we study the impact of hardware and software factors on the performance of realistic scientific workflows using WOWMON. In addition, we illustrate how its adaptive profiling technique helps reduce the volume of performance data at runtime. For evaluation we used an on-site cluster ACISS hosted at University of Oregon. The cluster includes 128 12-core Intel Xeon CPU X5650 2.67 GHz nodes running RedHat Enterprise release 6.6. All nodes are interconnected with a Voltaire Vantage 8500 10GigE network switch. The WOWMON library and related tools were compiled with the latest release of TAU [22], EVpath [5], and

ADIOS [1]. With respect to software configurations, *QueueDepth* for ADIOS is set to 20. We run one *LAMMPS* process per core unless specified otherwise.

4.1 Driving Scientific Workflows

We instrumented *LAMMPS* (Large Scale Atomic/Molecular Massively Parallel Simulator) [15] and its analytics *Bonds* and *Csym*, which are executed concurrently as the driving scientific workflow in the experiments. *LAMMPS* has been widely used for material science study. *Bonds* reads input from *LAMMPS* and performs all-nearest neighbor calculation to determine which atoms are bonds together. *Csym* then uses the output of *Bonds* to further determine whether there are deformation zones in the material. If they are detected, *Csym* continues to calculate the conditions under which a crack occurred. Therefore, its execution time and resource utilization may change over the whole period of simulation.

We manually select 12 metrics related to key functions of the *LAMMPS* workflow for tracing. As shown in Table 2, the metrics, such as *bonds_read_input* and *csym_output_results*, are related to I/Os for moving data from upstream applications to downstream analytics or to disks. And the metrics such as *bonds_compute_send* and *csym_compute_send* are related to computing kernels of *Bonds* and *Csym*, respectively. The other metrics are used to measure memory usage of a function. For example, *csym_compute_send_mem* measures the heap size of the function *csym_compute_send*.

4.2 The End-to-end Latency of the *LAMMPS* Workflow

The end-to-end latency of the *LAMMPS* workflow determines the time spent on analyzing physics of atoms using data generated by *LAMMPS*. We added 90 lines of code for source instrumentation of the metrics listed in Table 2. WOWMON can help capture transient behavior of workflows. To demonstrate this we show a breakdown of the end-to-end latency after completion of each simulation step during the execution of the LAMMPS workflow. The number of processes of *LAMMPS*, *Bonds*, and *Csym* are set to 128, 1, and 1, respectively.

From Figure 3, we observed that the end-to-end latency reached the maximum at the 42th step. According to its execution time breakdown, we found that during the execution of the workflow compute times of all the components did not change. In contrast, data transfer time is increased by 110% on average for both *LAMMPS* and *Csym*. With a careful study of this performance issues offline, we found the reason is that the ADIOS library uses in-memory queues to buffer output data of upstream applications for asynchronous execution. When the buffers became full after the 42th step, queuing time was dominant in the data transfer time as well as in the end-to-end latency.

4.3 Impact of a Workflow Mapping Policy

Given the same number of compute nodes, the policies of mapping processes of workflow components to nodes may significantly impact the end-to-end latency of scientific workflows. Using *LAMMPS* as an example, we demonstrate the performance difference between two mapping policies given 11 compute nodes. More specifically, $Policy_{CB}$ has *Csym* and *Bonds* placed on a dedicated node, while *LAMMPS* is executed on the other 10 nodes. There is no resource contention between *LAMMPS* and *Csym* or *LAMMPS* and *Bonds*. $Policy_{CL}$ has *Csym* and *LAMMPS* placed on the same nodes sharing memory and CPU resources. The above policies are different from the mapping policy used in the previous experiments in which processes use dedicated cores. In the experiment, *LAMMPS*, *Bonds* and *Csym* are configured with 128, 1,

Metric name	**Description**
bonds_read_input	Gives us a handle on throughput for reading data in *Bonds*.
bonds_list_output	Gives us a handle on throughput for moving data out of *Bonds*.
bonds_compute_send	Monitors the main computation function in *Bonds*. Most of its time is spent on building the adjacent-format output.
bonds_read_input_mem	The memory usage for executing bonds_read_input()
bonds_compute_send_mem	The memory usage for executing bonds_compute_send()
csym_read_input	Gives us a handle on throughput for reading data in *Csym*.
csym_compute_send	Monitors the main computation function in *Csym*. Most of its time is spent on converting input data.
csym_output_results	The corresponding function, which follows csym_output_results(), denotes the end of the workflow and where we end the latency measurement.
csym_read_input_mem	The memory usage for executing csym_read_input_mem()
csym_compute_send_mem	The memory usage for executing csym_compute_send_mem()
lammps_start_timer	The timer is triggered when the generated data is placed in buffers on LAMMPS end.
csym_stop_timer	The timer is triggered when the last analytic finishes.

Table 2: The selected metrics for the *LAMMPS* workflow

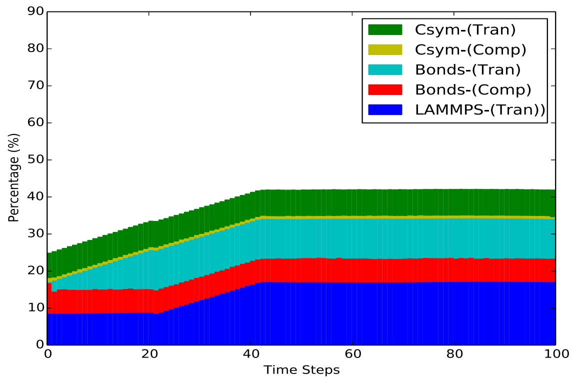

Figure 3: An execution time breakdown. *X-(Tran)* and *X-(Comp)* denote data transfer time and compute time of application X (Csym, Bonds, and LAMMPS), respectively.

and 1 processes, respectively. The end-to-end latency of each step with both $Policy_{CB}$ and $Policy_{CL}$ are shown in Figure 4. The average end-to-end latencies are 35.6s and 40.0s for

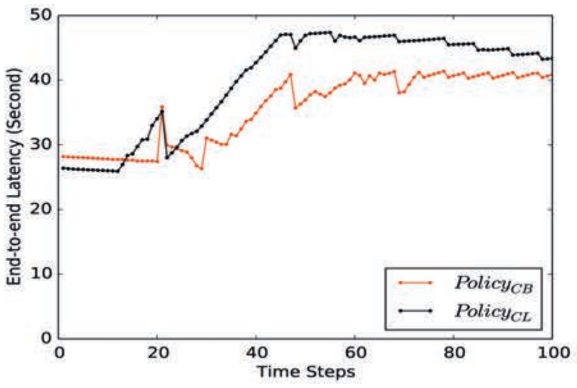

Figure 4: The end-to-end latency of the LAMMPS workflow with different policies of mapping processes of in-situ analytics to the compute nodes.

$Policy_{CB}$ and $Policy_{CL}$, respectively. One observation is that from the 1st to 14th step the end-to-end latency with $Policy_{CB}$ is higher. When the buffers for storing data for in-situ analysis are full after the 20th step, $Policy_{CB}$ has smaller latency than $Policy_{CL}$ since *LAMMPS* is more compute-intensive than *Bonds*. And extra interference between *LAMMPS* and *Csym* with $Policy_{CL}$ caused the higher latency.

4.4 Effectiveness of Self-adaptive Profiling

In this section, we study the correlation between selected profiling metrics and the performance of the *LAMMPS* workflow. In the experiment, the workflow is executed with 64 processes for *LAMMPS*, 2 processes for *Bonds*, and 1 process for *Csym*. Using the performance data collected from 1000 simulation steps as training data, the machine learning algorithm found one best metric subset including *bonds_read_input*, *csym_read_input*, and *csym_compute_send*. This result is consistent with our understanding that *Csym* is the bottleneck of the workflow for lack of parallelism in its design. We further measured the execution times of the functions corresponding to the selected metrics and found that they together account for about 46% of the average end-to-end latency. Figure 5 shows the results. This explains why they can be representative of the overall latency.

In addition, we evaluate the predictability of the models created based on either the selected metrics or the original metric set using a regression tree algorithm [23]. In detail, we created two machine learning models, $Model\_interest$ based on the three selected metrics and two timers, and $Model\_init$ based on the metrics in $MetricSet\_init$ as presented in Section 3.4. Both models use the selected metrics to predict the end-to-end latency of the workflow. 80% and 20% of the training set are used for creating a model and model validation, respectively. The validation results show that $Model_{interest}$ and $Model_{init}$ have relative absolute errors of 9.4% and 13.6%, respectively. This means $MetricSet_{interest}$ has better predictability than $Model_{init}$ because the accuracy of $Model\_interest$ is improved for less noises from the low-impact metrics in $MetricSet_{interest}$. Therefore, using $MetricSet_{interest}$ for monitoring can reduce the total amount of data transferred by 54% for the *LAMMPS* workflow without losing key metrics for performance debugging.

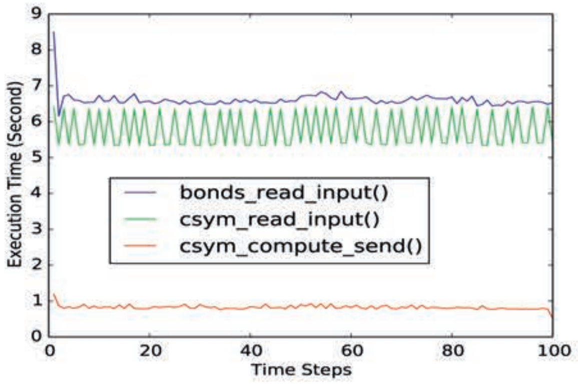

Figure 5: Execution times of the key functions of the *LAMMPS* workflow over 100 simulation time steps.

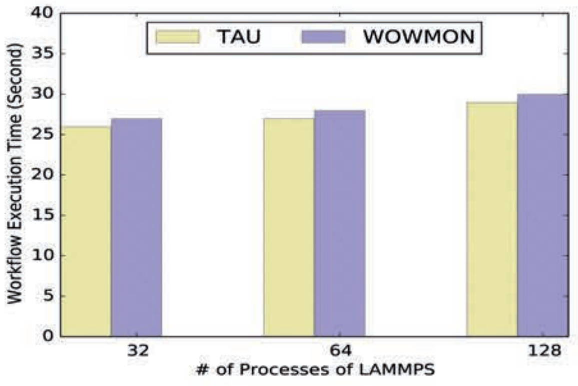

Figure 6: Total execution time of the LAMMPS workflow. TAU: applications run with the TAU profiler; WOWMON: applications run with WOWMON.

4.5 Overhead Analysis

There are three major sources of overhead in WOWMON. The first is profiling overhead, which has been well studied in many existing tools. WOWMON relies on these tools to provide scalable profiling services on various machines [16]. The second overhead is networking latency. We experimentally study the networking overhead of WOWMON compared to TAU, which is the default profiler in WOWMON. In the experiment, *LAMMPS* workflow is executed on ACISS with increased parallelism of *LAMMPS* from 32, 64, to 128 and one process for *Bonds*. As shown in Figure 6, the execution time with WOWMON is on average 7% more than that running with TAU alone. Its overhead grows sublinearly as the number of processes is increased. Third, WOWMON causes the memory overhead for maintaining performance data buffer at runtime. The size of the buffer is determined by the number of metrics in $MetricSet_{interest}$. For the *LAMMPS* workflow its buffer size is less than 20 KB with 128 processes.

5 Conclusion and Future Work

In the paper we present the design and implementation of WOWMON, a lightweight and powerful monitoring infrastructure, specifically designed for scientific workflows with in-situ analytics targeting large-scale computing platforms. It enables the online observation of adaptive workflow behavior. Evaluation with real scientific workflows shows that workflow performance can be affected by both software and hardware factors, such as the policy of process mapping and the size of memory buffer for in-situ processing. In addition, we show that WOWMON can automatically rank the correlation of each metric relative to the end-to-end latency of workflows using machine learning algorithms and reduce monitoring overhead with self-adaptive profiling/tracing. The overhead of WOWMON is 7% on average compared to the legacy profiling tools, making it well suit for online performance monitoring and analysis.

Future work for WOWMON will focus on the study of the effect of performance metrics across different runs with various input parameters to analytics (e.g., R_1 which determines how close atoms are for *Bonds*). In addition, with the help of WOWMON, we need to further understand the changes of data-driven behaviors of scientific workflows in emerging HPC platforms with deep memory hierarchies for caching in-situ output data.

6 Acknowledgements

We are grateful to the anonymous reviewers. We thank Jai Dayal and Matthew Wolf for their support on the LAMMPS workflow instrumentation. We also thank Wyatt Spear for many discussions during this work. This research was partially supported by DOE grant DE-SC0012381 funded by the Scientific Data Management, Analysis and Visualization (SDMAV) program and WSU Seed Grant.

References

[1] Adaptable IO System (ADIOS). https://www.olcf.ornl.gov/center-projects/adios/.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[3] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, 1998.

[4] Greg Eisenhauer, Matthew Wolf, Hasan Abbasi, and Karsten Schwan. Event-based systems: Opportunities and challenges at exascale. DEBS '09, pages 2:1–2:10. ACM, 2009.

[5] EVpath Library. http://svn.cc.gatech.edu/kaos/evpath/.

[6] Michael Gerndt, Andreas Schmidt, Martin Schulz, and R Wismuller. Performance analysis for teraflop computers: a distributed automatic approach. In *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 23–30. IEEE, 2002.

[7] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, and Je rey Vetter. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proceedings of the 5th Symposium of the Frontiers of Massively Parallel Computing*, pages 422–429, 1995.

[8] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[9] Mark A. Hall. Correlation-based feature selection for machine learning. Technical report, 1999.

[10] Kevin A. Huck, Allen D. Malony Sameer Shende, and Alan Morris. TAUg: Runtime Global Performance Data Access Using MPI. In *EuroPVM/MPI 2006*, 2006.

[11] Richard E. Korf. Depth-first iterative-deepening. *Artificial Intelligence*, 27(1):97 – 109, 1985.

[12] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30:2004, 2003.

[13] Barton P. Miller, Jonathan M. Cargille, R. Bruce Irvin, Krishna Kunchithapadam Mark D. Callaghan, Jeffrey K. Hollingsworth, Karen L. Karavanic, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *Computer*, 28:37–46, 1995.

[14] White Paper. Interconnect Analysis:10GigE and InfiniBand in High Performance Computing. *HPC Advisory Council Network of Expertise*.

[15] S. Plimpton, R. Pollock, and M. Stevens. Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations. In *Proc of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 2007.

[16] R.Bell, A.D. Malony, and S. Shende. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *EUROPAR' 03*.

[17] Reliable UDP networking library.
http://enet.bespin.org/.

[18] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The autopilot performance-directed adaptive control system. In *Future Generation Computer Systems*, pages 175–187, 1997.

[19] Philip C. Roth and Barton P. Miller. On-line automated performance diagnosis on thousands of processes. PPoPP '06.

[20] S. Shende and A. D. Malony. TAU: The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, pages 287–311, 2006.

[21] Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *Supercomputing 2002*, 2002.

[22] TAU Software.
https://www.cs.uoregon.edu/research/tau/downloads.php.

[23] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with cart models. pages 588–595, 2004.