MPI PERFORMANCE ENGINEERING WITH THE MPI TOOLS INFORMATION INTERFACE

by

SRINIVASAN RAMESH

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2018

THESIS APPROVAL PAGE

Student: Srinivasan Ramesh

Title: MPI Performance Engineering with the MPI Tools Information Interface

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Allen D. Malony	Chair
Sameer S. Shende	Core Member
and	
Sara Hodges	Interim Vice Provost and Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2018

THESIS ABSTRACT

Srinivasan Ramesh

Master of Science

Department of Computer and Information Science

June 2018

Title: MPI Performance Engineering with the MPI Tools Information Interface

The desire for high performance on scalable parallel systems is increasing the complexity and the need to tune MPI implementations. The MPI Tools Information Interface (MPI_T) introduced in the MPI 3.0 standard provides an opportunity for performance tools and external software to introspect and understand MPI runtime behavior at a deeper level to detect scalability issues. The interface also provides a mechanism to fine-tune the performance of the MPI library dynamically at runtime.

This thesis describes the motivation, design, and challenges involved in developing an MPI performance engineering infrastructure using MPI_T for two performance toolkits — the TAU Performance System®, and Caliper. I validate the design of the infrastructure for TAU by developing optimizations for production and synthetic applications. I show that the MPI_T runtime introspection mechanism in Caliper enables a meaningful analysis of performance data.

This thesis includes previously published co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Srinivasan Ramesh

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Birla Institute of Technology and Science (BITS), Pilani, RJ, India

DEGREES AWARDED:

Master of Science, Computer and Information Science, 2018, University of Oregon
Bachelor of Engineering (Hons.) Computer Science, 2014, BITS, Pilani

AREAS OF SPECIAL INTEREST:

High Performance Computing
Performance Optimization

PROFESSIONAL EXPERIENCE:

Graduate Research Fellow, Computer and Information Science, University of Oregon, 2016-Present

Computation Student Intern, Lawrence Livermore National Laboratory, Summer 2017

Research Assistant, Indian Institute of Science, 2015-2016

Software Engineer I, Amazon.com, 2014-2015

Intern, Optumsoft Research Pvt Ltd., 2013

GRANTS, AWARDS, AND HONORS:

Best Paper Award, EuroMPI, 2017

W.J. Cody Associate, Argonne National Laboratory, Summer 2018

Selected to attend the Argonne Training Program on Extreme-Scale Computing, 2018

Selected to attend the International High Perfomance Computing Summer School, 2017

Student Volunteer, Supercomputing, 2017

Student Travel Award, MVAPICH Users Group Meeting, 2017

Student Travel Award, MVAPICH Users Group Meeting, 2016

PUBLICATIONS:

Srinivasan Ramesh, Aurèle Mahéo, Sameer Shende, Allen D. Malony, Hari Subramoni, Amit Ruhela, and Dhabaleswar K. Panda. "MPI performance engineering with the MPI tool interface: the integration of MVAPICH and TAU." *Parallel Computing*. 2018.

Srinivasan Ramesh, Sathish Vadhiyar, Ravi S. Nanjundiah, and P. N. Vinayachandran. "Deep and Shallow Convections in Atmosphere Models on Intel Xeon Phi Coprocessor Systems." *HPCC*. 2017.

Srinivasan Ramesh, Aurèle Mahéo, Sameer Shende, Allen D. Malony, Hari Subramoni, and Dhabaleswar K. Panda. "MPI performance engineering with the MPI tool interface: the integration of MVAPICH and TAU." *Proceedings of the 24th European MPI Users Group Meeting (EuroMPI/USA)*. 2017.

ACKNOWLEDGEMENTS

The research leading up to this publication could not have been possible without the support of my family, advisors, close friends and well-wishers. My mother for constantly reminding me of the importance of prioritizing my health, my father for teaching me how to make difficult decisions and the art of financial planning, my advisor, Prof. Allen Malony for his constant encouragement, trust, and willingness to let me explore my research interests, my advisor, Dr. Sameer Shende for his invaluable technical advice, and my co-author and collaborator, Dr. Aurèle Mahéo for supporting and encouraging me to pursue my idealistic research ideas. I would like to thank my co-authors and collaborators at The Ohio State University — Dr. Hari Subramoni and Prof. Dhabaleswar K. (DK) Panda for providing me with helpful advice and research direction. I would like to thank my mentors at Lawrence Livermore National Laboratory — Dr. Martin Schulz, Dr. Tapasya Patki, and Dr. David Boehme for their guidance and patience. Lastly, I would like to thank all my close friends who have supported me in difficult times.

For my family, friends, and anyone who is passionate about High Performance Computing.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

This chapter includes co-authored material previously published in EuroMPI [1] and Parallel Computing [2]. These papers were the result of a collaboration with Aurèle Mahéo, Sameer Shende, Allen Malony, Hari Subramoni, Amit Ruhela, and Dhabaleswar (DK) Panda. I wrote the entire introduction section. My co-authors contributed by suggesting edits to improve the content.

The Message Passing Interface [3] remains the dominant programming model employed in scalable high-performance computing (HPC) applications. As a result, MPI performance engineering is worthwhile and plays a crucial role in improving the scalability of these applications. Traditionally, the first step in MPI performance engineering has been to profile the code to identify MPI operations that occupy a significant portion of runtime. MPI profiling is typically performed through the PMPI interface [4], wherein the performance profiler intercepts the MPI operation and performs the necessary timing operations within a wrapper function with the same name as the MPI operation. It then calls the corresponding name-shifted PMPI interface for this MPI operation. This technique generates accurate profiles without necessitating application code changes. The TAU Performance System® [5] is a popular tool that offers the user a comprehensive list of features to profile MPI applications through the PMPI profiling interface. PMPI profiling using TAU is performed transparently without modifying the application using runtime pre-loading of shared objects.

Although it plays a pivotal role in MPI performance engineering, using PMPI alone has some limitations:

– The profiler can only collect timing and message size data — it does not have access to MPI internal performance metrics that can help detect and explain performance issues.

– Profiling through the PMPI interface is mostly passive — it provides limited scope for interaction between the profiler and the MPI implementation.

Performance characteristics of underlying hardware are constantly evolving as HPC moves toward increasingly heterogeneous platforms. MPI implementations available today [6; 7; 8; 9] are complex software involving many modular components and offer the user a number of tunable environment variables that can affect performance. In such a setting, performance variations and scalability limitations can come from several sources. Detecting these performance limitations requires a more intimate understanding of MPI internals that cannot be elicited from the PMPI interface alone.

Tuning MPI library behavior through modification of environment variables presents a daunting challenge to the user — among the rich variety of variables on offer, the user may not be aware of the right setting to modify, or the optimal value for a setting. Further, tuning through MPI environment variables has a notable limitation — there is no way to fine-tune the MPI library at runtime. Runtime introspection and tuning are especially valuable to applications that display different behavior between phases, and one static setting of MPI parameters may not be optimal for the course of an entire run. In addition to this, each process may behave differently, and thus have a different optimal value for a given setting.

These complexities motivate the need for a performance measurement system such as TAU to play a more active role in the performance debugging and tuning process. With the introduction of the *MPI Tools Information Interface (MPI_T)* in

the MPI 3.0 standard, there is now a standardized mechanism through which MPI libraries and external performance tuning software can share information.

This document describes a software engineering infrastructure that enables an MPI implementation to interact with performance tuning software for the purpose of runtime introspection and tuning through the MPI_T interface. Specifically, I describe the design of an infrastructure to enable performance monitoring, runtime introspection, performance tuning, and recommendation generation of MPI applications using TAU, and an infrastructure designed for runtime introspection of MPI using Caliper [10].

Thesis Outline

While both the tools described in this document — TAU and Caliper offer performance measurement and analysis capabilities, they differ in terms of the data model used to store performance information, support for automatic program instrumentation, and the level of integration with other performance profiling and tracing toolkits. In terms of support for different performance engineering related tasks, TAU offers the user a much broader and *complete* set of features. Caliper is designed as a framework that relies on source-code annotation for data collection, and plugin based *services* that can be combined to provide custom features to the user.

Background and Related Work

In **Chapter 2** of the thesis, I discuss background material and related work in the areas of interfaces for runtime introspection, autotuning of MPI runtimes,

and tools that generate performance recommendations. I briefly introduce the software that are a part of this study.

Design of MPI_T support in TAU

In **Chapter 3**, I describe the design of the MPI_T support in TAU for performance introspection, monitoring, autotuning, and recommendation generation. Although the TAU MPI_T support is compliant with the MPI standard, it was designed in close collaboration with the MVAPICH2 [6] MPI implementation. Thus, this chapter describes the MVAPICH2-specific use cases that were used to motivate the design of the infrastructure. This chapter ends with a brief note on some of the challenges faced while implementing the design.

Design of MPI_T support in Caliper

In **Chapter 4**, I describe the design of the MPI_T support in Caliper for performance introspection. This work was performed at the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, during the summer of 2017. Caliper support was designed specifically for use with OpenMPI [7]. This shall conclude with a description of the unique design challenges encountered while working with OpenMPI.

Discussion

In **Chapter 5**, I present a discussion focusing on the differences between design and implementation of the MPI_T support in TAU and Caliper. Specifically, I focus on describing the differences in performance introspection alone, as Caliper at the moment does not have a GUI-based performance monitoring or plugin-like

tuning support for MPI_T. This discussion shall highlight the advantages and pitfalls of one design methodology over the other, and suggest areas for future work.

Conclusion and Future Work

In the concluding chapter, I shall present some directions for future work with regard to the MPI_T support in TAU. Specifically, I shall discuss my ongoing research efforts in exploring the potential benefits of enabling extremely fine-grained tuning of MPI point-to-point rendezvous protocols for non-blocking communication at runtime.

Coauthored Material

This thesis includes previously published co-authored material.

Chapter 2 and Chapter 3 include co-authored material previously published in the Proceedings of the 24th European MPI Users' Group Meeting (EuroMPI/USA 2017) [1], and the Journal of Parallel Computing [2].

CHAPTER II

BACKGROUND AND RELATED WORK

This chapter includes co-authored material previously published in EuroMPI [1] and Parallel Computing [2]. These papers were the result of a collaboration with Aurèle Mahéo, Sameer Shende, Allen Malony, Hari Subramoni, Amit Ruhela, and Dhabaleswar (DK) Panda. Hari Subramoni is the lead developer of the MVAPICH2 project and Sameer Shende leads the development of the TAU project. Aurèle Mahéo was instrumental in writing the related work section where I was minimally involved. This chapter describes the important software used in our study.

MPI Tools Information Interface

In order to address a lack of a standard mechanism to gain insights into, and to manipulate the internal behavior of MPI implementations, the MPI Forum introduced the MPI Tools Information Interface (MPI_T) in the MPI 3.0 standard [3]. The MPI_T interface provides a simple mechanism that allows MPI implementers to expose variables that represent a property, setting or performance measurement from within the implementation for use by tools, tuning frameworks, and other support libraries. The interface broadly defines access semantics of two variable types: *control* and *performance*. The former defines semantics to list, query and set control variables exposed by the underlying implementation. The latter defines semantics to gain insights into the state of MPI using counters, timing data, resource utilization data, and so on. Rich metadata information can be added to both kinds of variables.

Control variables (CVARs) are properties and configuration settings that are used to modify the behavior of the MPI implementation. A common example of such a control variable is the *Eager Limit* - the upper limit until which messages are sent using the Eager protocol. An MPI implementation may choose to export many environment variables as control variables through the MPI_T interface. Depending on what the variable represents, it may be set once before `MPI_Init` or may be changed dynamically at runtime. Further, the interface allows each process freedom to set its own value for the control variable provided the MPI implementation supports it. The MPI_T interface provides API's to read, write and query information about control variables and external tools can use these API's to discover information about the control variables supported.

Performance variables (PVARs) can represent internal counters and metrics that can be read, collected and analyzed by an external tool. An example of one such PVAR exported by MVAPICH2 is `mv2_vbuf_total_memory` which represents the total amount of memory used for internal communication buffers within the library. In a manner similar to CVARs, the interface specifies API's to query and access PVARs. MPI_T interface allows multiple in-flight performance sessions so it is possible for different tools to *plug into* MPI through this interface.

The MPI_T interface allows an MPI implementation to export any number of PVARs and CVARs, and it is the responsibility of the tool to discover these through appropriate API calls, and use them correctly. There are no fixed events or variables that MPI implementations must support - complete freedom is granted to the implementation in this regard.

Creating a Performance Session

In order to use MPI_T, a tool must first create a *performance session* and associate *handles* for the performance variables and control variables it wishes to read or write. Performance sessions allow the MPI library to distinguish between multiple tools/software modules that may be simultaneously querying the MPI_T interface.

Handle Allocation for PVARs and CVARs

Before a tool can read the value of a PVAR (CVAR), it must first allocate a *handle* for the PVAR (CVAR). The MPI_T interface specifies a function that allows a tool to know the number of PVARs (CVARs) exported by an MPI implementation at any given point in time. Two important points need to be kept in mind when allocating PVAR handles:

- Number of PVARs (CVARs) can change at runtime: The number of PVARs (CVARs) exported by the library can change at any point during runtime. Typically, MPI libraries export additional PVARs (CVARs) after `MPI_Init`. A tool must be able to support and account for dynamic expansion and invalidation of PVARs (CVARs) at runtime as and when they become available and fall out of scope respectively.

- PVARs (CVARs) can be bound to MPI objects: The `MPI_pvar_get_info` function returns the *bind* type for the PVAR. The idea here is that PVARs (CVARs) can be *associated* with a specific object such as a communicator or message. As a result, there can be multiple handles allocated for a PVAR (CVAR) at any given index. These handles must be allocated appropriately

depending on the bind type. Additional detail regarding bind types shall be provided in **Chapter 4**.

It is worth noting that the only MPI implementation that exports PVARs and CVARs bound to MPI objects is OpenMPI. As of the time of publishing this document, all of the other popular MPI libraries that support MPI_T — namely MPICH, MVAPICH2, and Intel MPI export PVARs and CVARs that have a bind type `none`. As we shall see in **Chapter 4**, the need to support variables bound to MPI objects significantly enhances the complexity of the design on the tool side.

Software

This work targets the development of an integrated software infrastructure that enables the use of MPI_T for performance introspection and online tuning. Here I describe the functionality of the key software components used in this study.

TAU Performance System®

TAU [5] is a comprehensive performance analysis toolkit that offers capabilities to instrument and measure scalable parallel applications on a variety of HPC architecture. TAU[1] supports standard programming models including MPI and OpenMP. It can profile and trace MPI applications through the PMPI interface either by linking in the TAU library or through library interposition. TAU includes a tool for parallel profile analysis (ParaProf), performance data mining (PerfExplorer), and performance experiment management (TAUdb).

[1] http://tau.uoregon.edu

Caliper

Caliper [10][2] is a general purpose application introspection system developed at Lawrence Livermore National Laboratory that relies on source-code annotation for performance data collection. It provides users with a measurement API and a flexible key-value data format for storing performance information, along with a host of *services* that be combined to offer the user a customized performance measurement and analysis solution. Unlike TAU, Caliper does not offer GUI-based tools for performance analysis. It is designed as a framework rather than a comprehensive toolkit, and it can be integrated with other existing performance tools through Caliper's tool API to leverage their capabilities.

MVAPICH2

MVAPICH2 [6][3] is a cutting-edge open source MPI implementation for high-end computing systems that is based on the MPI 3.1 standard. MVAPICH2 currently supports InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE networking technologies. It offers the user a number of tunable environment parameters and has GPU and MIC optimized versions available.

BEACON

BEACON (Backplane for Event and Control Notification) [11] is a communication infrastructure, originally part of the Argo project [12]. BEACON provides interfaces for sharing event information in a distributed manner, through nodes and enclaves - a group of nodes. It relies on a Publish/Subscribe paradigm,

[2]https://github.com/LLNL/Caliper

[3]http://mvapich.cse.ohio-state.edu/download/mvapich/mv2/mvapich2-2.3rc2.tar.gz

and encompasses backplane end-points (called BEEPs) which are in charge of detecting and generating information to be propagated throughout the system. Other BEEPs subscribe to this information and can generate appropriate actions. Events are exchanged between publishers and subscribers through user-defined topics. Examples of such topics are power, memory footprint and CPU frequency. These interfaces allow BEACON to be called and used by external components such as performance tools for exchanging information. BEACON also includes a modular GUI named PYCOOLR that provides support for dynamic observation of intercepted events. PYCOOLR subscribes to these events by using the BEACON API and is able to display their content during application runtime. Through the GUI, the user can select at runtime the events that represent the performance metrics he wants to observe, and the GUI plots the selected events on the fly.

Related Work

The existing body of research on MPI performance engineering techniques has revolved around a few common themes. These include design and usage of interfaces similar in spirit to MPI_T, user interactions with performance tools for the purpose of tuning, and automatic tuning of MPI runtimes. We describe some contributions addressing these areas below.

Interfaces for Runtime Introspection

Throughout MPIs history, it has always been of interest to application developers to observe the inner workings of the MPI implementation. Early attempts to open up an implementation for introspection gained some traction in the tools community. PERUSE [13] allows observation of internal mechanisms of

MPI libraries by defining callbacks related to certain events, illustrated by specific use cases. For instance, the user can have a detailed look at the behavior of MPI libraries during point-to-point communications. This interface was implemented inside OpenMPI. But it failed to be adopted as a standard by the MPI community, mainly due to a potential mismatch between MPI events proposed by PERUSE and some MPI implementations.

With the advent of the MPI_T interface, Islam et al. introduce Gyan [14], using MPI_T to enable runtime introspection. Gyan intercepts the call to `MPI_Init` through the PMPI interface, initializes MPI_T and starts a PVAR monitoring session to track PVARs specified by the user through an environment variable. If no PVAR is specified, Gyan tracks all PVARs exported by the MPI implementation. Gyan intercepts `MPI_Finalize` through PMPI, reads the values of all performance variables being tracked through the MPI_T interface, and displays statistics for these PVARs. Notably, Gyan collects the values of PVARs only once at `MPI_Finalize`, while our infrastructure supports tracking of PVARs at regular intervals during the application run, in addition to providing online monitoring and autotuning capabilities.

Performance Recommendations

Other contributions, focusing on tuning MPI configuration parameters, provide performance recommendations to the users. MPI Advisor [15; 16] starts from the idea that application developers do not necessarily have sufficient knowledge of MPI library design. This tool is able to characterize predominant communication behavior of MPI applications and gives recommendations on how the runtime can be tuned. It addresses the following parameter categories: (i)

point-to-point protocols (*Eager* vs *Rendezvous*), (ii) collective communication algorithms, (iii) MPI task-to-cores mapping, and (iv) Infiniband transport protocol. The execution of MPI Advisor comprises three phases: data collection, analysis, and recommendations. MPI Advisor uses mpiP [17] to collect application profiles and related information such as message size and produce recommendations to tune MPI_T CVARs. It requires only one application run on the target machine to produce recommendations. While our recommendation engine is similar in functionality to MPI Advisor, our infrastructure leverages TAU's profiling capabilities to give us access to more detailed application performance information. This enables us to implement more sophisticated recommendation policies. The focus of our work is a plugin infrastructure that enables recommendation generation as one of many possible usage scenarios, and not a sole outcome.

Another tool, OPTO (The Open Tool for Parameter Optimization) [18], aids the optimization of OpenMPI library by systematically testing a large number of combinations of the input parameters. Based on the measurements performed on MPI benchmarks, the tool is able to output the best attribute combinations.

Autotuning of MPI Runtimes

Some tools introduce autotuning capabilities of MPI applications by deducing best configuration parameters, involving different techniques for searching. Periscope and its extensions [19; 20], part of the AutoTune project, provide capabilities of performance analysis and autotuning of MPI applications, by studying runtime parameters. Starting from different parameter configurations specified by the user, the tool generates a search space. It then searches for the best values, by using different strategies involving heuristics such as evolutionary

algorithms and genetic algorithms. Based on measurements obtained by running experiments, the tool finds the best configuration parameters. ATune [21] uses machine learning techniques to automatically tune parameters of the MPI runtime. The tool runs MPI benchmarks and applications on a target platform to predict parameter values, via a training phase. To the best of our knowledge, there exists no prior work of autotuning MPI runtimes using the MPI_T interface.

Policy Engine for Performance Tuning

Outside the scope of MPI, APEX [22] was developed as a part of the XPRESS project, which includes a parallel programming model named OpenX, and a runtime implementing this model, HPX. APEX provides runtime introspection and includes a policy engine introduced as a core feature: the policies are rules deciding the outcome based on observed states of APEX. These rules can thus change the behavior of the runtime — such as changing task granularity, triggering data movements or repartitioning.

Summary

This chapter has described the background concepts, prior work, and software used in our study. The next chapter shall focus on describing the design of the MPI_T support in TAU.

CHAPTER III

DESIGN OF MPI_T SUPPORT IN TAU

This chapter includes co-authored material previously published in EuroMPI [1] and Parallel Computing [2]. These papers were the result of a collaboration with Aurèle Mahéo, Sameer Shende, Allen Malony, Hari Subramoni, Amit Ruhela, and Dhabaleswar (DK) Panda. I implemented the plugin support in TAU and integrated BEACON and TAU. Hari Subramoni is the lead developer of the MVAPICH2 project and Sameer Shende leads the TAU project. Aurèle Mahéo implemented the GUI support in PYCOOLR for performance monitoring. Sameer Shende designed and implemented the initial version of the MPI_T based performance introspection in TAU and provided critical guidance in designing the plugin support in TAU. I designed and implemented the experiments described in this chapter. For the autotuning experiments on AmberMD and 3DStencil, I received support from Hari Subramoni and Amit Ruhela.

The existence of MPI_T provides an opportunity to link together the components above. However, each component must be extended to interact through the MPI_T interface, as well as in concert with each other. Although applicable to any standard-compliant MPI implementation, the design of the MPI_T support in TAU was performed in close collaboration with the MVAPICH2 MPI library. Below, we describe the design approach for MVAPICH2 and TAU integration to enable runtime introspection, performance tuning, and recommendation generation. Figure 1 depicts the infrastructure architecture and component interactions. We then present sample usage scenarios for this infrastructure.

Enhancing MPI_T Support in MVAPICH2

MVAPICH2 exports a wide range of performance and control variables through the MPI_T interface. A performance variable represents an internal metric or counter, and setting a control variable may alter the behavior of the library. Current support for MPI_T variables in MVAPICH2 broadly fall under the following categories:

Monitoring and Modifying Collective Algorithms

For collective operations such as `MPI_Bcast` and `MPI_Allreduce`, there are a variety of algorithms available and the right algorithm to use depends on a number of parameters such as system metrics (bandwidth, latency), the number of processes communicating and the message size. MVAPICH2 exports CVARs that can be used to determine the collective algorithm based on the message size. It also supports PVARs that monitor the number of times a certain collective algorithm is invoked.

Monitoring and Controlling Usage of Virtual Buffers

Virtual Buffers (VBUFs) are used in MVAPICH2 to temporarily store messages in transit between two processes. The use of virtual buffers can offer significant performance improvement to applications performing heavy point-to-point communication, such as stencil-based codes. MVAPICH2 offers a number of PVARs that monitor the current usage level, availability of free VBUFs in different VBUF pools, maximum usage levels, and the number of allocated VBUFs

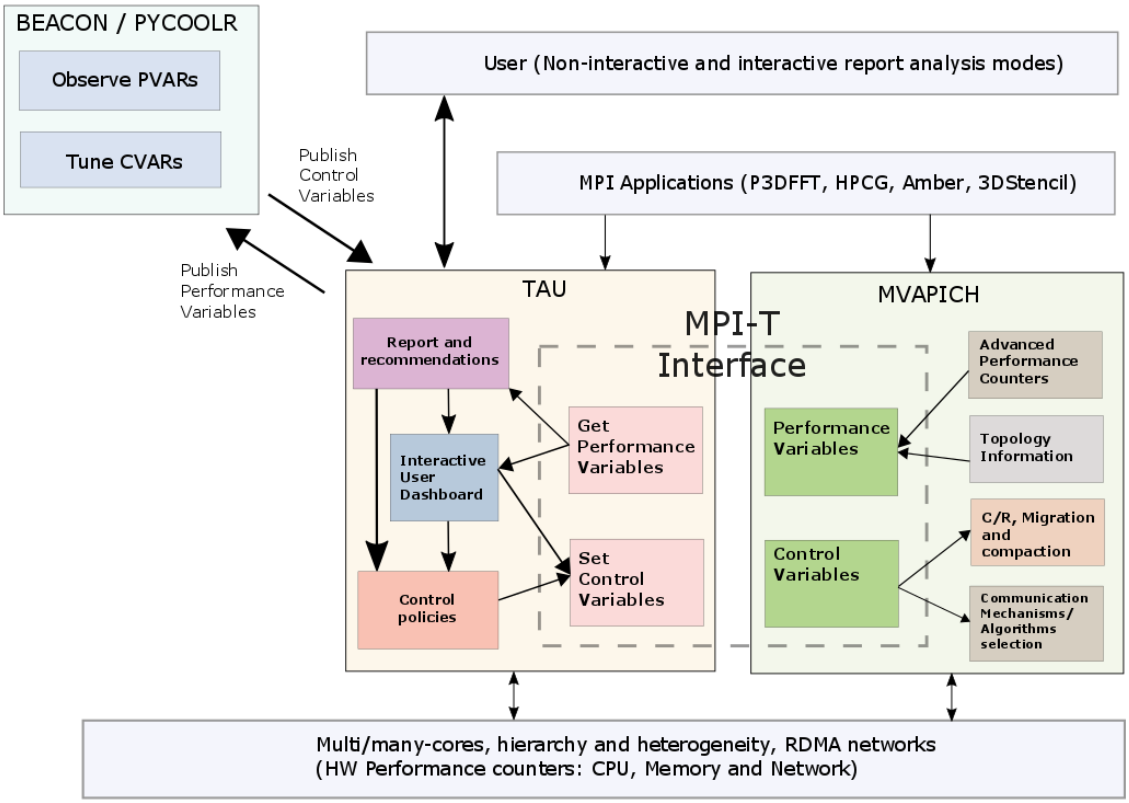

FIGURE 1. Integrated MVAPICH2 and TAU infrastructure based on MPI_T

at process level granularity. Accordingly, it exposes CVARs that modify how MVAPICH2 allocates and frees these VBUFs at runtime.

Enabling Runtime Instrospection and Online Monitoring

MPI_T makes it possible to inquire about the state of the underlying MPI implementation through the query of performance variables. While it is the prerogative of the MPI implementation what PVARs are published, the tool must be extended to use MPI_T for access. Similarly, control variables are defined by the MPI implementation but set by the tool using MPI_T. Below we discuss how this is done in TAU to realize introspection and tuning.

Gathering Performance Data

TAU has been extended to support the gathering of performance data exposed through the MPI_T interface. Each tool that is interested in querying MPI_T must first register a *performance session* with the interface. This object allows the MPI library to store separate contexts and differentiate between multiple tools/components that are simultaneously querying the MPI_T interface. Along with a *performance session*, a tool must also allocate *handles* for all the performance variables that it wishes to read/write. Within TAU, the task of allocating the global (per-process) performance session and handles for PVARs is carried out inside the TAU tool initialization routine. However, this design has a caveat — an MPI library can export additional PVARs during runtime as they become available through dynamic loading. A tool must accordingly allocate handles for these additional PVARs if it wishes to read them. TAU currently does

not support this — we are restricted to reading PVARs that are exported at TAU initialization. We plan to support the dynamic use case in a future release.

TAU can use sampling to collect performance variables periodically. When an application is profiled with TAU's MPI_T capabilities enabled, an interrupt is triggered at regular intervals. Inside the signal handler for the `SIGALRM` signal, the MPI_T interface is queried and the values of *all* the performance variables exported are stored at process level granularity. TAU registers internal *atomic user events* for each of these performance variables, and every time an event is triggered (while querying the MPI_T interface), the running average, minimum value, the maximum value, and other basic statistics are calculated and available to the user at the end of the profiling run. These statistics carry meaning only for PVARs that represent `COUNTERS` or `TIMERS`. Thus, we define TAU user events to store and analyze PVARs for these two classes. The MPI_T interface allows MPI libraries to export PVARs from a rich variety of classes — timers, counters, watermarks, state information, and so on. MVAPICH2 and TAU have been primarily designed to support PVARs from the `TIMER` or `COUNTER` classes. As part of future work, we plan to export a richer variety of PVAR classes and design appropriate methods for storage and analysis of each of these classes.

TAU also provides an application-level API to query all exported PVARs as and when the application desires (i.e., in a synchronous manner). However, we have chosen not to demonstrate this method of sampling PVARs in our experiments.

Online Monitoring

Runtime introspection naturally extends to online monitoring where certain performance variables are made viewable during execution. Figure 2 depicts the

interaction between TAU and BEACON to enable online monitoring of PVARs through the PYCOOLR GUI.

To interface TAU and BEACON, TAU defines a BEACON *topic* for performance variables and publishes PVAR data collected at runtime to this topic. Any software component interested in monitoring PVARs can then subscribe to this topic and receive live updates for all performance variables exported by the MPI implementation.

To monitor PVARs on PYCOOLR, the PYCOOLR GUI acts as a subscriber to the PVAR topic — thus it receives updated values for all PVARs from TAU's sampling-based measurement module. The GUI has been extended to offer the user the ability to select only those PVARs that he/she is interested in monitoring — this is a useful feature as an MPI library can export 100's of PVARs, not all of which may interest the user. The GUI plots the values for the selected PVARs at runtime as and when it receives them through BEACON.

Viewing Performance Data

ParaProf is the TAU component that allows the user to view and analyze the collected performance profile data post-execution. This profile information is collected on a per-thread or a per-process level, depending on whether or not threads were used in the application. ParaProf has existing support for the analysis of *interval events* as well as *atomic user events*. Interval events are used to capture information such as the total execution time spent inside various application routines. Atomic user events are used to store information such as hardware counter values.

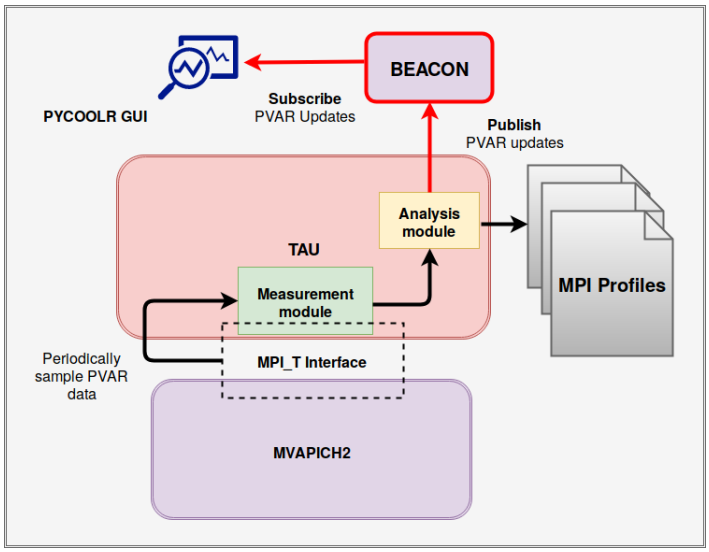

FIGURE 2. Online monitoring with BEACON/PYCOOLR

PVARs are treated as atomic user events. ParaProf's existing support for analyzing atomic user events has been leveraged to display PVAR data for each process. Performance variables collected from the MPI_T interface during execution are displayed on ParaProf as events that include markers indicating high variability.

Runtime Tuning through MPI_T

Complementary to providing an API for runtime introspection, the MPI_T interface also enables a mechanism to modify the behavior of the MPI library through control variables. MPI implementations can define control variables for configuration, performance, or debugging purposes. MPI libraries may implicitly restrict the semantics of *when* CVARs can be set — some may be set only once before `MPI_Init`, and others may be set anytime during execution. Further, there may be restrictions on whether or not CVARs are allowed to have different values for different processes — this decision is left entirely up to the MPI library.

Therefore, a tool or a user interacting with the MPI_T interface for the purpose of tuning the MPI library must be aware of the particular semantics associated with the CVARs of interest.

User-Guided Tuning through PYCOOLR

Our infrastructure provides users the ability to fine-tune the MPI library by setting CVARs at runtime. As depicted in Figure 3, we use the BEACON backplane communication infrastructure to enable user-guided tuning. TAU and BEACON interface with each other in a bi-directional fashion. Aside from acting as a publisher of PVAR data, TAU is a subscriber to a BEACON topic used for communicating CVAR updates. The PYCOOLR GUI has been extended to enable the user to set new values for multiple CVARs at runtime — Figure 4 displays a screenshot of the PYCOOLR window that enables this functionality.

Together with the online monitoring support provided by PYCOOLR, this user-guided tuning infrastructure can enable a user to experiment with different settings for CVARs and note their effects on selected PVARs or other performance metrics. We must note that this infrastructure has one significant limitation — the value that the user sets for a CVAR is *uniformly* applied across MPI processes. In other words, each MPI process receives the same value for the CVAR — this may not be ideal, as it is likely that each process displays a different behavior and thus may have a different optimal value for a given setting. We argue that this infrastructure is nevertheless useful in the experimentation phase, wherein the user is trying to determine the CVAR that is important for a given situation/application.

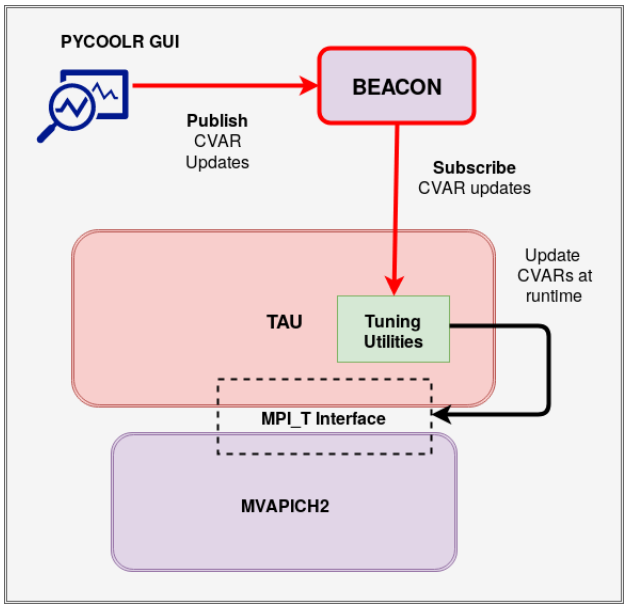

FIGURE 3. User-guided tuning with BEACON/PYCOOLR

Plugin Infrastructure in TAU

TAU is a comprehensive software suite that is comprised of well-separated components providing instrumentation, measurement and analysis capabilities. Our vision for performance engineering of MPI applications involves a *more active* involvement of TAU in monitoring, debugging and tuning behavior *at runtime*. The MPI_T interface provides tools an opportunity to realize this vision.

Recall that the MPI_T interface allows MPI implementations complete freedom in defining their own PVARs and CVARs to export. However, this freedom comes with a cost to tool writers for MPI_T — each MPI implementation will require its own custom tuning and re-configuration logic. From a software infrastructure development standpoint, it would be preferable to design a framework that will allow multiple such customized autotuning logic to co-exist *outside of core tool logic*, and be appropriately loaded depending on the MPI library being used. With this motivation in mind, we have added support for a

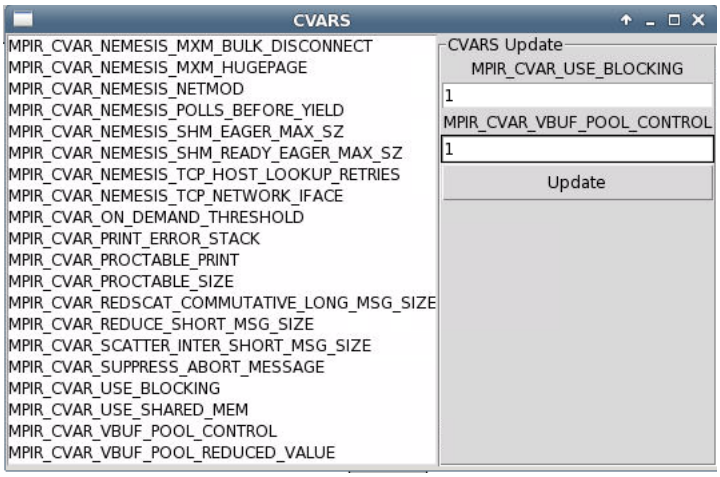

FIGURE 4. Screenshot of PYCOOLR window to update CVARs

generic plugin infrastructure in TAU that can be used to develop and load custom logic for a variety of performance engineering needs. The latest version of TAU supports this plugin infrastructure.

Design Overview

In the current design, plugins are `C/C++` modules that are built into separate shared libraries (Dynamic Shared Objects). The path to the directory containing the plugins is specified using the environment variable `TAU_PLUGINS_PATH`. The list of plugins to be dynamically loaded at runtime is specified using the environment variable `TAU_PLUGINS` separated by a colon as a delimiter.

In keeping with the general design of plugin frameworks, the TAU plugin system has the following stages:

– **Initialization**: This is invoked during TAU library initialization. During this phase, TAU's plugin manager reads the environment variables `TAU_PLUGINS_PATH` and `TAU_PLUGINS` and loads the plugins in the order

specified by `TAU_PLUGINS`. Each plugin ***must*** implement a function called `Tau_plugin_init_func`. Inside this function, it can register callbacks for a subset of *plugin events* it is interested in. Note that each plugin may register callbacks for more than one event. The plugin manager maintains an *ordered* list of active plugins for each event supported.

- **Event Callback Invocation**: We define some salient plugin events in TAU that could be interesting or useful from a performance engineering standpoint. These events are discussed in detail in the section that follows. When these plugin events occur during execution of an application instrumented with TAU, the plugin manager invokes the registered callbacks for the specific event in the order in which the corresponding plugins were loaded. Each event that is supported has a specific, typed data object associated with it. When the event occurs, this data object is populated and sent as a parameter to the plugin callback.
- **Finalize Phase**: When TAU is done generating the profiles for the application, the plugins are unloaded, and all the auxiliary memory resources allocated by the plugin manager are freed.

Plugin Events Supported

Plugin events are entry points into the plugin code that performs a custom task. Currently, TAU defines and supports the following events:

- `TAU_PLUGIN_EVENT_FUNCTION_REGISTRATION`: TAU creates and registers a `FunctionInfo` object for all functions it instruments and tracks. This event marks the end of the registration phase for the `FunctionInfo` object that was created.

- `TAU_PLUGIN_EVENT_ATOMIC_EVENT_REGISTRATION`: TAU defines *atomic events* to track PAPI counters, PVARs and other entities which do not follow interval event semantics. This plugin event marks the end of the registration phase for the atomic event and is triggered when the atomic event is created. In the context of our MPI_T infrastructure, this plugin event is triggered once for every PVAR that is exported by the MPI library.

- `TAU_PLUGIN_EVENT_ATOMIC_EVENT_TRIGGER`: When the value of an atomic event is updated, this event is triggered. This plugin event is triggered once for each PVAR, every time the MPI_T interface is queried.

- `TAU_PLUGIN_EVENT_INTERRUPT_TRIGGER`: TAU's sampling subsystem relies on installing an interrupt handler for the `SIGALRM` signal, and performs the sampling within this interrupt handler. When TAU is used with its sampling capabilities turned on, this plugin event is triggered within TAU's interrupt handler (10 seconds is the default interrupt interval).

- `TAU_PLUGIN_EVENT_END_OF_EXECUTION`: When TAU has finished creating and writing the profile files for the application, this plugin event is triggered.

There may be other supported events added in future releases.

Use Case: Filter Plugin to Disable Instrumentation at Runtime

To demonstrate a sample usage scenario for the plugin architecture, we have created a plugin that filters out instrumented functions from being profiled at runtime, based on a user-provided selective instrumentation file. This situation arises when the application has been instrumented using either the compiler or TAU's source instrumentation tool — the Program Database Toolkit (PDT) [23].

PDT works by parsing the input source file to detect function definitions and function call sites, and automatically adds the TAU instrumentation API calls to these sites. The user may want to prevent certain *automatically instrumented functions* from being profiled — these functions may be frequently invoked but not have a significant impact on overall runtime. They may pollute the generated profiles and more importantly, add to the measurement overheads without providing any real benefit. From a profiling standpoint, there is solid motivation to provide a mechanism that allows such functions to be excluded from profiling.

We use our plugin infrastructure to provide this functionality — our filter plugin registers a callback for the `TAU_PLUGIN_EVENT_FUNCTION_REGISTRATION` plugin event. Recall that this event is triggered once for every function instrumented by TAU. Within the callback for the

`TAU_PLUGIN_EVENT_FUNCTION_REGISTRATION` event, we read a user-provided selective instrumentation file that contains a list of functions to be excluded from profiling. The data object for this plugin event contains the function name information. If there is a match between the function being registered and the list of function names in the selective instrumentation file, we set the profile group for the function to be `TAU_DISABLE`, effectively switching off profiling for this function.

Plugins for Autotuning

Figure 5 depicts TAU plugins in the context of our MPI_T infrastructure. As discussed earlier, TAU samples PVAR data from the MPI_T interface inside a signal handler for the `SIGALRM` signal. TAU can use this collected PVAR data to perform an autotuning decision inside the signal handler — this is realized through plugins that install callbacks for the `TAU_PLUGIN_EVENT_INTERRUPT_TRIGGER` event.

This event is triggered every time TAU samples the MPI_T interface, and the registered plugin callbacks are invoked. Inside the callback, the plugin has access to all the PVAR data collected and performs a *runtime autotuning decision* that may result in updated values for *one or more* CVARs (knobs). Plugins can make use of core TAU modules to interact with the MPI_T interface to update CVAR values.

Note that the plugin infrastructure allows the user to specify more than one plugin — this feature can be utilized to load multiple autotuning policies, each of which is built into a separate shared library. While plugins use common functionality defined inside TAU to read or write to the MPI_T interface, the autotuning logic itself is custom to each plugin — in the future, we plan to support a high-level infrastructure to express autotuning policies that reduce duplicated code across plugins. Our starting point for developing autotuning policies relies on users with background or offline knowledge about specific domains, applications, and libraries.

Plugins for Recommendations

We take advantage of the plugin mechanism to develop performance recommendations for the user. MPI libraries can export a large number of control variables — many of which are also environment variables whose default settings may not always be optimal for a given application/situation. Moreover, the user may not even be aware of the existence of certain settings or MPI implementation-specific features that can improve performance. A profiling tool such as TAU is

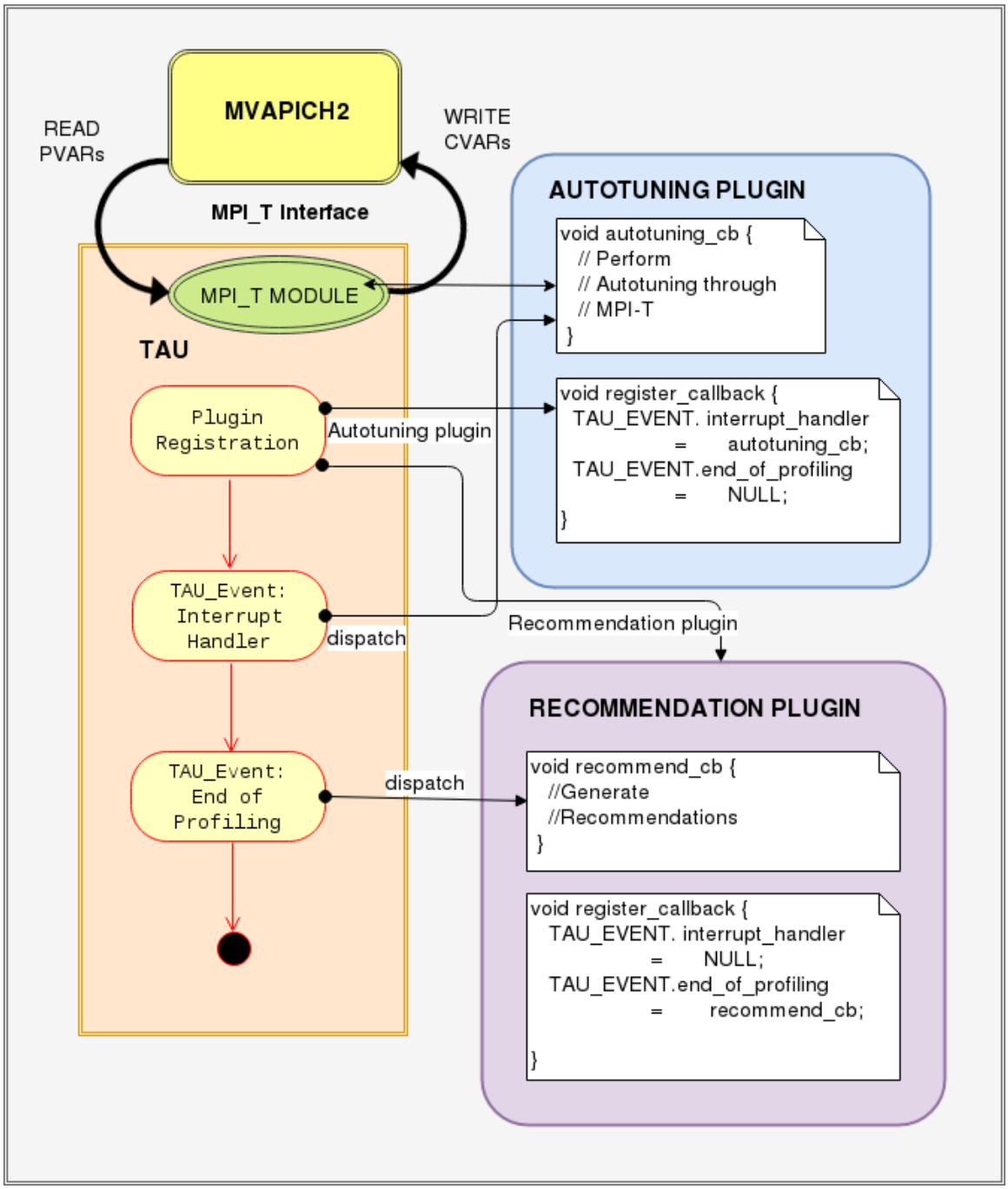

FIGURE 5. Plugin infrastructure

in an ideal position to fill this gap with the MPI_T interface acting as an enabling mechanism.

Performance data gathered by TAU through the MPI_T and PMPI interface can be analyzed by a recommendation plugin to provide useful hints to the user at the end of the application execution. Recommendation plugins register callbacks for the `TAU_PLUGIN_EVENT_END_OF_EXECUTION` event that is triggered when TAU has finished collecting and writing profile information. Currently, TAU supports the generation of recommendations as part of the metadata that is associated with each process. This metadata is available for viewing on ParaProf.

Target Applications

AmberMD

AmberMD [24] is a popular software package that consists of tools to carry out molecular dynamics simulations. A core component is the molecular dynamics engine, `pmemd`, which comes in two flavors: serial and an MPI parallel version. We focus on improving the performance of molecular dynamics simulations that use the parallel MPI version of `pmemd`. A substantial portion of the total runtime is attributed to MPI communication routines, and among MPI routines, calls to `MPI_Wait` dominate in terms of contribution to runtime. However, in terms of number of MPI calls made, `MPI_Isend` and `MPI_Irecv` dominate. The use of non-blocking sends and receives explicitly allows the opportunity for a greater communication-computation overlap.

SNAP

SNAP [25] is a proxy application from the Los Alamos National Laboratory that is designed to mimic the performance characteristics of PARTISN [26]. PARTISN is a neutral particle transport application that solves the linear Boltzmann transport equation for determining the number of neutral particles in a multi-dimensional phase space. SNAP is considered to be an updated version of the Sweep3D [27] proxy application and can be executed on hybrid architectures. SNAP heavily relies on point-to-point communication, and the size of messages transferred is a function of the number of spatial cells per MPI process, number of angles per octant, and number of energy groups.

Specifically, a bulk of the point-to-point communication is implemented as a combination of `MPI_Isend`/`MPI_Waitall` on the sender side, and `MPI_Recv` on the receiver side. This explicitly allows the opportunity for communication-computation overlap on the sender side.

3DStencil

We designed a simple synthetic stencil application that performs non-blocking point-to-point communication in a cartesian grid topology. In between issuing the non-blocking sends and receives and waiting for the communication to complete, the application performs arbitrary computation for a period of time that is roughly equal to the end-to-end time for pure communication alone. The goal is to evaluate the degree of communication-computation overlap. In an ideal scenario of 100% overlap, the computation would complete at the same time as communication, so that no additional time is spent in waiting for the non-blocking

communication requests to complete. For the purposes of this experiment, point-to-point communication involves messages of an arbitrarily high, but fixed size.

MiniAMR

MiniAMR is a mini-app that is a part of the Mantevo [28] software suite. As the name suggests, it involves adaptive mesh refinement and uses 3D Stencil computation. MiniAMR is a memory bound application, and communication time is dominated by `MPI_Wait` for point-to-point routines involving small messages (1-2 KB range) and `MPI_Allreduce`. The `MPI_Allreduce` call involves messages of a constant, small size (8 bytes) making it latency sensitive. This call is part of the check-summing routine and increasing the check-summing frequency or the number of stages per timestep impacts the scalability of this routine and thus the application.

Usage Scenarios

MPI_T in combination with the TAU plugin architecture makes it possible to do powerful operations that would be difficult to realize otherwise. The following describes the design of a recommendation to enable hardware offloading of collectives, and an autotuning policy to free unused MPI internal buffers using MPI_T. These policies are implemented using plugins.

Recommendation Use Case: Hardware Offloading of Collectives

MVAPICH2 now supports offloading of `MPI_Allreduce` to network hardware using the SHArP [29] protocol. Hardware offloading is mainly beneficial to applications where communication is sensitive to latency. As the `MPI_Allreduce`

call in MiniAMR involves messages of 8 bytes, it is a prime candidate to benefit from hardware offloading.

During the profiling phase, TAU collects statistics about the average message size involved in `MPI_Allreduce` operation. It also collects the time spent within `MPI_Allreduce` versus the overall application time. If the message size is below a certain threshold and the percentage of total runtime spent within `MPI_Allreduce` is above a certain threshold, through ParaProf, TAU recommends the user to set the CVAR `MPIR_CVAR_ENABLE_SHARP` for subsequent runs. Note that this recommendation policy was implemented using plugins. The same infrastructure can be used to support multiple recommendation policies.

Autotuning Use Case: Freeing Unused Buffers

MVAPICH2 uses internal communication buffers (VBUFs) to temporarily hold messages that are yet to be transferred to the receiver in point-to-point communications. There are multiple VBUF pools which vary in size of the VBUF. At runtime, MVAPICH2 performs a match based on the size of the message and accordingly selects a VBUF pool to use. Specifically, these VBUFs are used when MVAPICH2 chooses to send the message in an *Eager* manner to reduce communication latency. Typically, short messages are sent using the *Eager* protocol, and longer messages are sent using the *Rendezvous* protocol, which does not involve the use of VBUFs. The primary scalability issue with using *Eager* protocol is excessive memory consumption that can potentially lead to an application crash.

Depending on the pattern of message sizes involved in point-to-point communication, the usage level of these VBUF pools can vary with time and

between processes. It can be the case that the application makes scarce use of VBUFs, or uses VBUFs only from one pool (3DStencil is one such use case). In such a scenario, unused VBUFs represent wasted memory resource. There could be significant memory savings in freeing these unused VBUFs.

For this use case, specific CVARs include:

- `MPIR_CVAR_IBA_EAGER_THRESHOLD`: The value of this CVAR represents the message size above which MVAPICH2 uses the *Rendezvous* protocol for message transfer in point-to-point communication. Below this message size, MVAPICH2 uses the *Eager* protocol

- `MPIR_CVAR_VBUF_TOTAL_SIZE`: The size of a single VBUF. For best results, this should have the same value as `MPIR_CVAR_IBA_EAGER_THRESHOLD`

- `MPIR_CVAR_VBUF_POOL_CONTROL`: Boolean value that specifies if MVAPICH2 should try to free unused VBUFs at runtime. By default, MVAPICH2 will try to free from any available pool if this variable is set

- `MPIR_CVAR_VBUF_POOL_REDUCED_VALUE`: This CVAR specifies the lower limit to which MVAPICH2 can reduce the number of VBUFs. This is an array, and each index represents the corresponding VBUF pool. This CVAR takes effect only if pool control is enabled. This CVAR allows more fine-grained control over freeing of VBUFs, potentially reducing unnecessary allocations and freeing of VBUFs, if the usage pattern is known in advance

Correspondingly, PVARs of interest include:

- `mv2_vbuf_allocated_array`: Array that represents the number of VBUFs allocated in a pool specified by an index

– `mv2_vbuf_max_use_array`: Array that represents the maximum number of VBUFs that are actually used in a given pool specified by an index

– `mv2_total_vbuf_memory`: Total VBUF memory (in bytes) used for the specified process across all pools

Autotuning Policy

When we increase the value of the *Eager* limit specified by `MPIR_CVAR_IBA_EAGER_THRESHOLD`, there is an opportunity for increased overlap between communication and computation as larger messages are sent eagerly. As a result, the overall execution time for the application may reduce. Figure 6 is an enlarged Vampir [30] summary process timeline view for one iteration of the 3DStencil application before applying the Eager optimization. Figure 7 is a Vampir summary process timeline view for one iteration of the 3DStencil application after applying the Eager optimization. The timeline view focuses on the phase of the iteration where there is an explicit opportunity for communication-computation overlap through the use of non-blocking sends and receives. The X-axis represents time and the Y-axis represents the percentage of MPI processes inside user code (green) and MPI code (red) respectively at any given instant in time — larger areas of green indicates a higher amount of useful work (computation) performed by processes as a result of a larger communication-computation overlap.

Figure 7 shows the effect of an increased Eager threshold — a 20% increase in the number of MPI processes inside user code during the phase where communication is overlapped with computation. This increase is due to the fact that less time is spent waiting for the non-blocking calls to complete at the `MPI_Wait` barrier. With a larger eager threshold, the MPI library can advance

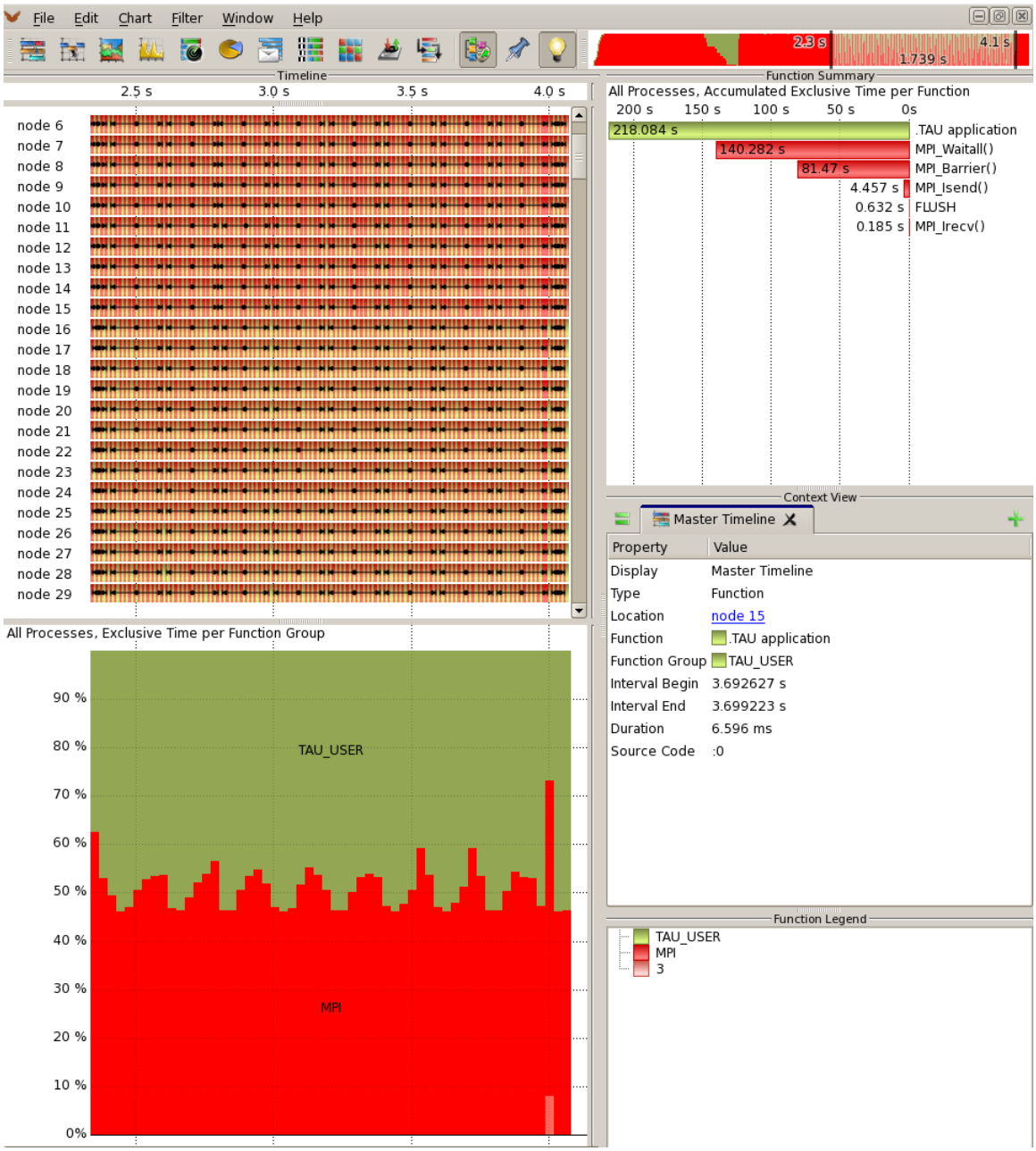

FIGURE 6. 3DStencil: Vampir process timeline view before Eager tuning

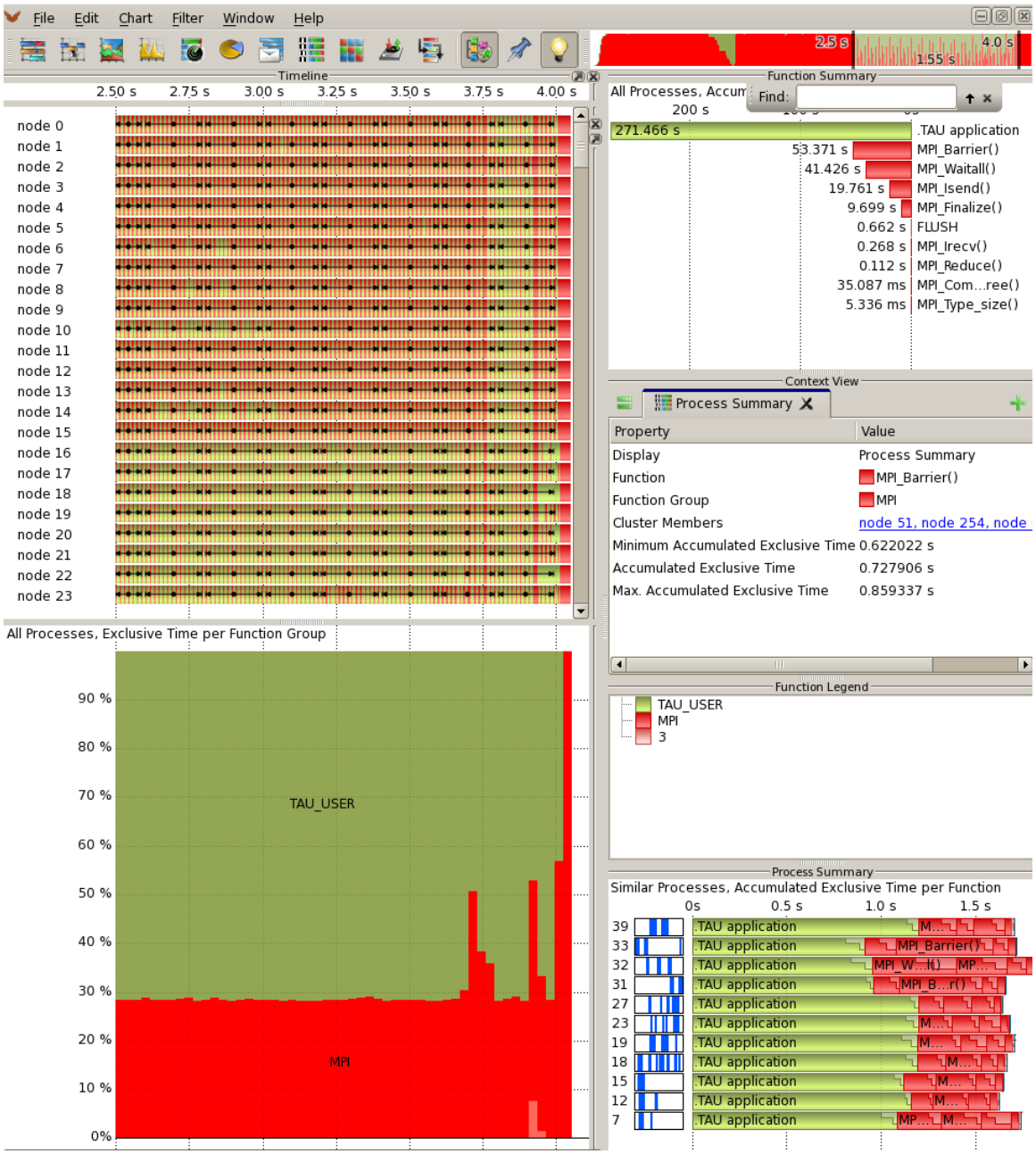

FIGURE 7. 3DStencil: Vampir process timeline view after Eager tuning

communication in the background while the sending process is busy performing the computation. The extreme right edges of Figure 7 are to be ignored as they represent the phase where the application is performing pure communication.

Increasing the Eager limit may have the following two distinct effects:

- Larger VBUFs may need to be allocated. Note that this does not mean that *more* VBUFs are allocated — it only means that the size of each individual VBUF in the affected pool has increased in order to hold larger messages. Recall that MVAPICH2 has four VBUF pools — the VBUFs from different pools vary in only their size.

- As a result of the increased Eager limit, larger messages would be transferred through the Eager protocol instead of the Rendezvous protocol. Depending on the communication characteristics of the application, this may lead to increased usage of VBUFs from one or more VBUF pools. If there is a shortage of VBUFs in a given pool, MVAPICH2 may need to allocate additional VBUFs.

A combination of these two factors may lead to an increase in the total VBUF memory usage inside MVAPICH2. Figure 8 is a PYCOOLR screenshot illustrating this increase in total VBUF memory usage (across all four pools) for AmberMD application when the Eager threshold is raised. We see a similar increase in total VBUF memory usage for the 3DStencil application as well. The X-axis represents time and the Y-axis represents memory in bytes with 10^7 as the multiplier. Each red dot represents the instantaneous `mv2_total_vbuf_memory` (in bytes) for one MPI process. If MPI processes have the same VBUF memory usage at any point in time, then the red dots would overlap. From Figure 8, it is evident that

there are two classes of processes — one with a VBUF memory usage of roughly 3 MB (before Eager tuning), and another with a VBUF memory usage level of roughly 6 MB (before Eager tuning). The eager threshold is raised by setting the CVAR `MPIR_CVAR_IBA_EAGER_THRESHOLD` and `MPIR_CVAR_VBUF_SIZE` statically, during `MPI_Init`. Figure 8 shows that the `mv2_total_vbuf_memory` increases to approximately 12 MB for the processes with a lower VBUF memory usage, and approximately 23 MB for the class of processes with a higher VBUF memory usage.

While one pool sees an increase in VBUF usage, it is possible that other VBUF pools may have unused VBUFs that can be freed to partially offset this increased memory usage inside MPI. In applications such as 3DStencil or AmberMD where the message size is fixed or in a known range, VBUFs from only one pool is used. In such a scenario, freeing unused VBUFs from other pools leads to significant memory savings. The usage levels of VBUF pools would vary from one application to another depending on the particular characteristics of the point-to-point communication.

MPI_T offers a mechanism to monitor pool usage at runtime. Our autotuning policy implemented as a plugin monitors the difference between the *array* PVARs `mv2_vbuf_allocated_array` and `mv2_vbuf_max_use_array` — each pool has a unique ID, and this unique ID is used to index into these two arrays. The difference between these two quantities at a given pool index represents the quantity of wasted memory resource in that pool. If this difference breaches a user-defined threshold for at least one pool, the autotuning policy sets the CVAR `MPIR_CVAR_VBUF_POOL_CONTROL` to enable MVAPICH2 to free any unused VBUFs. In order to enable more fine-grained control over freeing of unused VBUFs, MVAPICH2 exports an *array* CVAR `MPIR_CVAR_VBUF_POOL_REDUCED_VALUE`.

This CVAR is used to communicate the minimum number of VBUFs that must be available in each pool after enabling pool control. When the threshold for unused VBUFs is breached for at least one pool, the autotuning plugin enables pool control and simultaneously sets this array CVAR to be equal to the array PVAR `mv2_vbuf_max_use_array` — this is a heuristic that is employed to determine the new values for the various VBUF pool sizes. If on the other hand, this threshold is not breached for *any* pool, the TAU autotuning plugin unsets the CVAR `MPIR_CVAR_VBUF_POOL_CONTROL` effectively turning off further attempts to free unused VBUFs by MVAPICH2 until the next time the threshold is breached.

Alternatively, both these CVARs can be set at runtime through the PYCOOLR GUI as well — however, the advantage of using an autotuning plugin for this purpose is that these values can be set individually and independently for different processes. It can also be more responsive without incurring the delay of communicating with the PYCOOLR GUI. Setting different CVAR values for different processes is not possible through the PYCOOLR GUI.

Figure 9 depicts the decrease in `mv2_total_vbuf_memory` for AmberMD when only `MPIR_CVAR_VBUF_POOL_CONTROL` is enabled through the PYCOOLR GUI, instructing MPI to free any unused VBUFs. Note that the autotuning plugin is not employed here. The CVAR for pool control is enabled at around the 150-second mark, and at this point, the VBUF memory usage levels drop as a result of unused VBUFs being freed.

Experiments

In this section, we present the results obtained from applying the autotuning and recommendation policies to our target applications — AmberMD, SNAP,

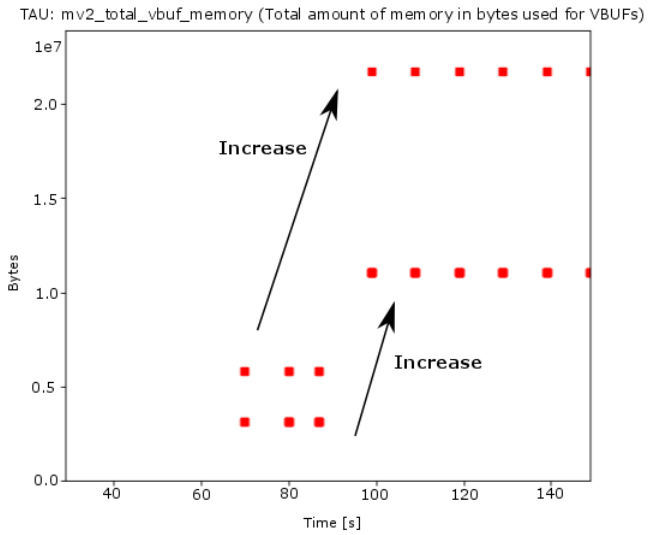

FIGURE 8. PYCOOLR: Total VBUF memory with higher Eager threshold

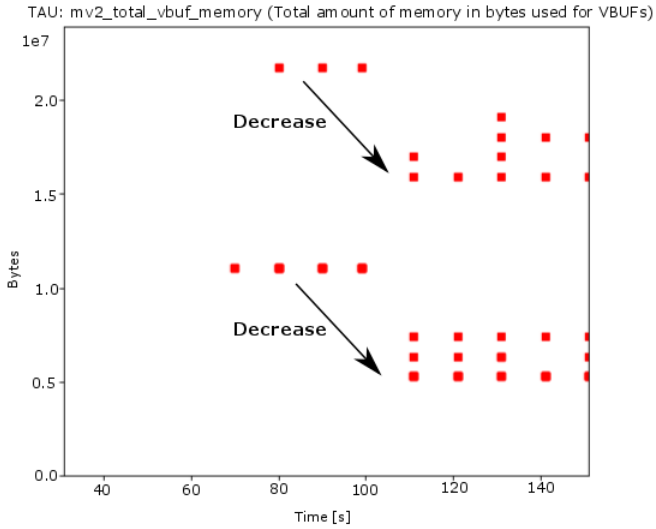

FIGURE 9. PYCOOLR: Total VBUF memory after freeing unused VBUFs

3DStencil, and MiniAMR. We describe results from a study of overheads involved in enabling MPI_T in MVAPICH2 and TAU.

Experimental Setup

Our experiments with AmberMD, SNAP, and 3DStencil were performed on Stampede, a 6400 node Infiniband cluster at the Texas Advanced Computing Center [31]. Each regular Stampede compute node has two Xeon E5-2680 8-core "Sandy Bridge" processors and one first-generation Intel Xeon Phi SE10P KNC MIC. We chose to run all our experiments using pure MPI on the Xeon host with 16 MPI processes on a node (1 per core) with `MV2_ENABLE_AFFINITY` turned on so that MPI tasks were pinned to CPU cores. For SNAP, we used a total of 64 nodes (at 16 processes per node, a total of 1024 processes). For our experiments with AmberMD and 3DStencil, we used a total of 32 nodes (at 16 processes per node, a total of 512 processes).

Experiments with MiniAMR and those involving a study of sampling overheads using 3DStencil were performed on the ri2 Infiniband cluster at The Ohio State University. Each compute node on ri2 has two 14-core Intel Xeon E5-2680 v4 processors. The HCA on all nodes in the cluster is the Mellanox CX-4 100 Gigabit adapter. The OFED version used is MLNX_OFED_LINUX-4.1-1.0.2.0 and the Linux kernel version is 3.10.0-327.10.1.el7.x86_64. We ran all our experiments using pure MPI on Intel Xeon hosts with 28 MPI processes on a node (1 per core) and pinned the MPI processes. We used a total of 2-16 nodes (at 28 processes per node, a total of 56 to 448 processes) for our experiments with 3DStencil, and a total of 8 nodes (at 28 processes per node, a total of 224 processes) for experiments with MiniAMR.

Results

Amber

Table 1 summarizes the results of modifying the *Eager* threshold and applying the runtime autotuning policy for AmberMD. The threshold is set statically right after MPI initialization, using `MPIR_CVAR_IBA_EAGER_THRESHOLD`. We noted that increasing the *Eager* threshold from the MVAPICH2 default value to 64000 bytes had the effect of reducing application runtime by 19.2%. This was achieved at the cost of increasing the total VBUF memory across all processes by 320%. Please note that the total VBUF memory usage reported here is the *average* value across the number of times that this metric was sampled (once every 10 seconds). The third row shows results of applying the user-defined policy of freeing unused VBUFs at runtime, on top of the *Eager* threshold optimization. We saw a sizeable reduction in total VBUF memory used while the runtime remained unaffected.

SNAP

SNAP application relies heavily on point-to-point communication, and the message sizes involved in communication depend on a number of input factors. We followed the recommended ranges for these input factors:

TABLE 1. AmberMD: Impact of Eager threshold and autotuning

Run	Number of Processes	Eager Threshold (Bytes)	Timesteps	Runtime (secs)	Total VBUF Memory(KB)
Default	512	MVAPICH2 Default	8,000	166	4,796,067
Eager	512	64,000	8,000	134	15,408,619
TAU autotuning	512	64,000	8,000	134	15,240,073

– Number of angles per octant (nang) was set to 50

– Number of energy groups (ng) was set to 150

– Number of spatial cells per MPI rank was set to 1200

– Scattering order (nmom) was set to 4

We gathered the message sizes involved in MPI communication. Table 2 lists the five MPI functions that account for the highest aggregate time spent. `MPI_Recv` and `MPI_Waitall` together account for nearly 17% of total application time or 60% of MPI time. Table 3 lists the message sizes involved in various MPI routines. It is evident that the bulk of messages are point-to-point messages with a message size of roughly 18,300 bytes.

The fact that the application spends a lot of its communication time inside `MPI_Recv` (callsite ID 5 in Table 2) and `MPI_Waitall` (callsite ID 16 in Table 2) suggests that the receiver in the point-to-point communication is generally late as compared to the posting of the corresponding `MPI_Isend` (callsite ID 1 in Table 2) operation. As a result of the relatively large message size of 18KB involved in this case, the data is transferred using the Rendezvous protocol *after* the receive is posted — in this specific context, this data transfer happens when the sender reaches the `MPI_Waitall` call. Even though there is an opportunity for communication-computation overlap through the use of non-blocking routines, no overlap actually happens in the application because of the conditions necessary for the transfer of large messages using the Rendezvous protocol.

By increasing both the inter-node and intra-node Eager threshold to 20KB, the transfer of these point-to-point messages is initiated when the sender posts the `MPI_Isend` operation. As a result, the application sees an increase in

TABLE 2. SNAP: Aggregate time inside various MPI functions

MPI routine name	Callsite ID	Portion of Application Runtime (%)	Portion of MPI Time (%)
MPI_Recv	4	13.31	47.50
MPI_Barrier	5	5.20	18.55
MPI_Allreduce	7	3.84	13.72
MPI_Waitall	16	3.09	11.02
MPI_Isend	1	1.21	4.33

TABLE 3. SNAP: Average sent message sizes from various MPI functions

MPI routine name	Callsite ID	Count	Average Message Size (Bytes)
MPI_Isend	1	114348672	18300
MPI_Allreduce	7	25600	1200
MPI_Send	18	2400	1920
MPI_Bcast	11	1024	120
MPI_Bcast	15	1024	120

communication-computation overlap, and this manifests itself as a reduction in overall application runtime. The second row of Table 4 summarizes this improvement in performance with 1024 processes — we note a reduction of 10.7% in application runtime when increasing the Eager threshold to 20 KB. However, increasing the Eager threshold also meant that the total VBUF memory usage across all processes went up by 12%.

The TAU autotuning plugin ensures that VBUFs from unused pools are freed to offset this increase in total VBUF memory usage. The third row of Table 4 summarizes the reduction in total VBUF memory usage when the TAU autotuning plugin is enabled. It is important to note that the plugin does not disturb application runtime even at this scale.

TABLE 4. SNAP: Impact of Eager threshold and autotuning

Run	Number of Processes	Eager Threshold (Bytes)	Runtime (secs)	Total VBUF Memory(KB)
Default	1024	MVAPICH2 Default	47.3	3,322,067
Eager	1024	20,000	42.2	3,787,050
TAU autotuning	1024	20,000	42.9	2,063,421

3DStencil

Table 5 summarizes the results of these experiments with our synthetic 3DStencil code. We designed the application in such a way that non-blocking point-to-point communications involve messages of an arbitrarily high, but fixed size. We measured the communication-computation overlap achieved. The first row describes results for the default run, where a very low communication-computation ratio of 6.0% was achieved as messages are sent using the *Rendezvous* protocol. The reason for setting a high, but fixed value for message size was to ensure that only VBUFs from one pool are utilized. In a manner similar to AmberMD, this application benefited from an increased value for the *Eager* threshold. The communication-computation ratio went up from 4.6% to 79.9% and as a result, there was a corresponding drop in application runtime by 26.2%. However, the total VBUF memory utilized went up by 1.6 times as compared to the default setup. We noted significant benefits in implementing the runtime autotuning policy of freeing unused VBUFs, although it still was 1.49 times of the original.

It is important to note that the actual amount of memory freed through the autotuning logic depends on the usage levels of various pools. With both AmberMD and 3DStencil, the message sizes involved in the communication were relatively large — as a result, smaller size VBUFs were freed.

TABLE 5. 3DStencil: Impact of Eager threshold and autotuning

Run	Number of Processes	Message Size (Bytes)	Eager Threshold (Bytes)	Overlap (%)	Runtime (secs)	Total VBUF Memory(KB)
Default	512	32,768	MVAPICH2 Default	4.6	198.1	3,112,302
Eager	512	32,768	33,000	79.9	146.6	4,893,712
TAU autotuning	512	32,768	33,000	80.0	146.4	4,644,691

MiniAMR

Table 6 summarizes the results of enabling SHArP for MiniAMR. Both the default and optimized runs were performed under similar conditions, with increased values for check-summing frequency and stages per timesteps to better demonstrate the potential benefits of enabling hardware offloading of collectives. Under these conditions, we saw an improvement of 4.6% in runtime when enabling SHArP on 8 nodes.

TABLE 6. MiniAMR: Impact of hardware offloading on application runtime

Run	Number of Processes	Runtime (secs)
Default	224	648
SHArP enabled	224	618

Overhead in Enabling MPI_T

MVAPICH2 does not enable tracking of MPI_T PVARs by default. This feature is enabled by configuring MVAPICH2 with the `--enable-mpit-pvars` flag. Enabling and tracking PVARs has a cost associated with it, and we sought to quantify this cost for small-scale experiments using our infrastructure. Recall that when TAU is configured to track PVARs, TAU samples PVARs at regular intervals — the default value for the sampling interval is 10 seconds. TAU reads

every PVAR exposed by the MPI implementation — this implies that the overhead with sampling is directly proportional to the number of the PVARs exported.

Using a version of MVAPICH2 with MPI_T disabled as the baseline, we set up experiments to measure the following overheads:

- Overhead of enabling MPI_T within MVAPICH2
- Overhead of sampling PVARs at regular intervals using TAU

In our first set of experiments, we sought to quantify the cost of enabling MPI_T within MVAPICH2, and the cost of sampling at the default rate inside TAU (once every 10 seconds). We measured the execution time for the 3DStencil application on the ri2 cluster — with 28 MPI processes per node, we ran experiments using 2 to 16 nodes. Each experiment was repeated 5 times, and the average execution time was calculated. The results of these experiments are depicted in Figure 10.

At small-scale, we see negligible overheads with our infrastructure. The execution times for MVAPICH2 configured with MPI_T are nearly identical to the execution times for the baseline. When sampling at the default rate of once every 10 seconds, TAU's sampling system does not seem to add any noticeable overhead to the execution time — we see a maximum of 4.7% overheads when using 2 nodes. With other node counts, the overheads are low enough to be indistinguishable from the run-to-run variability that is dependent on non-deterministic factors.

In our second set of experiments, we studied how runtime is affected by sampling more frequently from the MPI_T interface. We measured the execution time of the 3DStencil application on the ri2 cluster using 16 nodes (at 28 processes per node, a total of 448 processes). Each experiment was repeated 5 times, and the average execution time was calculated — we have not presented the error

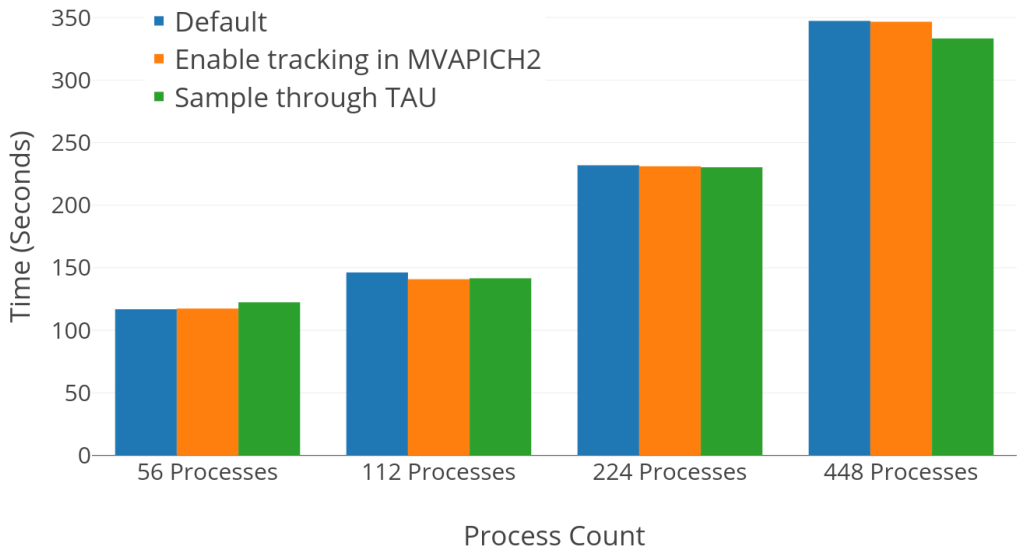

FIGURE 10. Overhead in enabling MPI_T for 3DStencil

bars with the results because there was negligible variation between runs. Figure 11 shows that the overheads are negligible even when sampling at a rate of once every second. In summary, the overall runtime for the 3DStencil application is not affected noticeably when the sampling rate is increased. Although it may not be the most suitable method for all usage scenarios, this study suggests that sampling provides a low-overhead solution for tracking PVARs. These experiments suggest that our infrastructure is likely to scale to large node counts. Overhead studies with large node counts will be part of our future work.

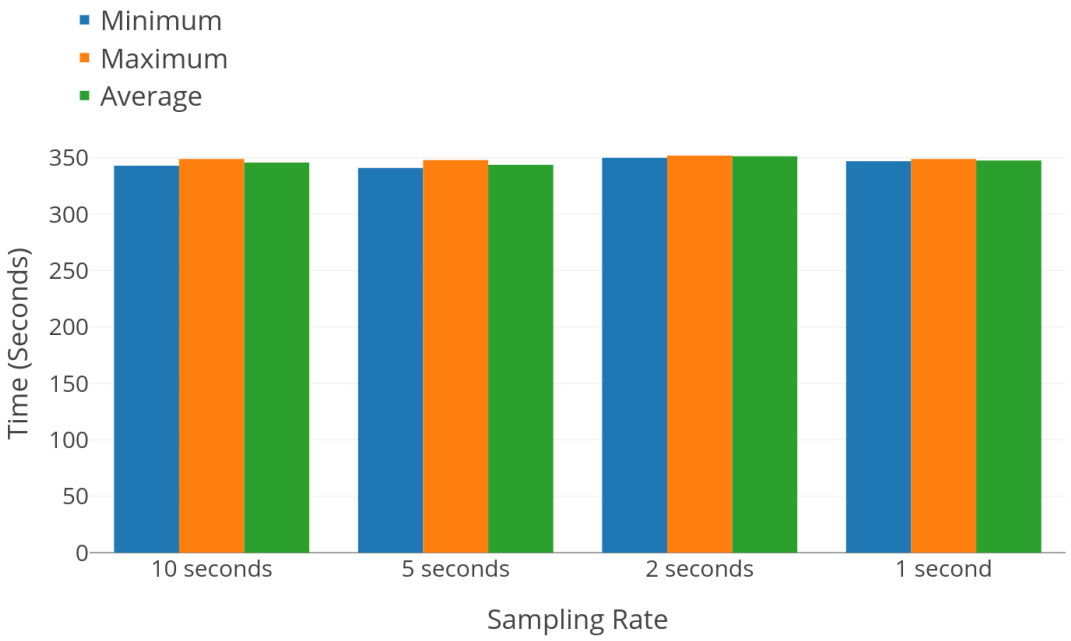

FIGURE 11. Effect of MPI_T sampling frequency on overhead for 3DStencil

Implementation Challenges and Issues

Deadlock Inside Signal Handler

The sampling mechanism inside TAU works by installing a signal handler to the `SIGALRM` signal. In order to prevent deadlocks, it is vital that all the callpaths inside the signal handler (TAU routines) are signal-safe, and only make signal-safe library calls. Our initial implementation of the autotuning plugin made free use of the `malloc` and `calloc` library calls — these are not signal-safe.

As it turned out, when running the TAU autotuning plugin with a large (300 or more) number of MPI processes, some processes were interrupted while inside a call to `malloc` from within the MVAPICH2 MPI library. As the plugin itself invoked `malloc`, this led to a deadlock on the `heap-lock`. This issue was mitigated by using TAU's custom memory manager to request for heap memory. A valuable lesson was learned in the process of detecting and resolving this issue — a tool

using interrupt-based sampling must make no assumption about the use of signal-unsafe routines inside the MPI library. In order to ensure proper functionality, tool writers' must always pessimistically assume that the library makes use of signal-unsafe routines, and design around this assumption.

Supporting Dynamic Expansion of MPI_T variables

Recall that the number of PVARs and CVARs exported by the MPI library can increase (or decrease) at runtime. Supporting the case where the number decreases at runtime is trivial — the tool just invalidates the PVARs (or CVARs) at the specific indices, and doesn't query the interface for variables at these indices.

Supporting the scenario where the number of variables *increases* at runtime, however, is a more tricky task. As discussed earlier in this chapter, TAU maintains a *user event* for every PVAR exported. In the default setup where TAU samples the MPI_T interface inside a signal handler, TAU becomes aware of the increase in the number of variables only inside the signal handler. As a result, it needs to allocate the PVAR (CVAR) handles for the additional PVARs (CVARs), and create a TAU *user event* for each of these additional PVARs (CVARs). Unfortunately, a call to the MPI_T routine that allocates PVAR (CVAR) handles invokes a `malloc` call inside MVAPICH2.

This leads to the same deadlock problem discussed previously. Moreover, at the time of detecting this issue, it was discovered that a lot of routines that lie in the callpath for the creation of a TAU *user event* were signal-unsafe. As a result, it was decided that TAU would not support dynamic expansion of MPI_T variables for the time being.

The lesson here is — sampling-based techniques for MPI_T are bound to be limited by the use of signal-safe routines, and this tends to increase the complexity of the tool implementation. For instance, one workaround for TAU would have been to store the fact that the number of MPI_T variables has increased (state information), and then invoke the handle allocation routines from within the PMPI wrapper for the next MPI call. In order for this solution to work perfectly, this check must be performed inside the PMPI wrapper for every MPI routine, thereby increasing the overall overheads for the TAU PMPI wrapper when MPI_T is enabled. This solution was abandoned owing to the potential performance risks, and the manual work involved in implementing it.

Summary

This chapter has described the design and implementation of the MPI performance engineering architecture in TAU. We have also described the usage scenarios for this architecture and validated the design by performing experiments on synthetic and production scientific applications. The next chapter shall describe the MPI_T based performance introspection support in Caliper.

CHAPTER IV

DESIGN OF MPI_T SUPPORT IN CALIPER

Caliper is an application introspection tool that relies on source code annotations to collect information and perform profiling related tasks. I shall first provide a basic overview of relevant Caliper concepts before describing the MPI_T support in Caliper.

Caliper Concepts

Caliper API

Caliper provides an application level API that acts as the portal for carrying out performance measurements. Caliper also provides high-level annotation macros that are user-friendly. The basic idea behind the source-code annotation API is to associate performance measurements with user-defined, high-level *context information*. These source code annotations act as hooks for background processing. Caliper is built into a library and linked into the application. Figure 12[1] is an example of a Caliper-annotated `C++` source code.

Attributes: Caliper's Building Blocks

Caliper provides a generic *key-value* data model for storing performance data of all kinds. Caliper *attributes* are the basic elements of the Caliper data model. The keys need to have a unique name and a type. They can also optionally have properties which determine how the attributes get processed. An example would be

[1]Image taken from: https://llnl.github.io/Caliper

```
int main(int argc, char* argv[])
{
    // Mark this function
    CALI_CXX_MARK_FUNCTION;

    // Mark the "intialization" phase
    CALI_MARK_BEGIN("initialization");
    int count = 4;
    double t = 0.0, delta_t = 1e-6;
    CALI_MARK_END("initialization");

    // Mark the loop
    CALI_CXX_MARK_LOOP_BEGIN(mainloop, "main loop");

    for (int i = 0; i < count; ++i) {
        // Mark each loop iteration
        CALI_CXX_MARK_LOOP_ITERATION(mainloop, i);

        // A Caliper snapshot taken at this point will contain
        // { "function"="main", "loop"="main loop", "iteration#main loop"=<i> }

        // ...
    }

    CALI_CXX_MARK_LOOP_END(mainloop);
}
```

FIGURE 12. Caliper annotated source code

an attribute to track PAPI counters or an attribute to track the total time spent inside a routine or code section.

Among all the properties that an attribute can have, the most important property in the context of MPI_T is the `AS_VALUE` property. Attributes with the `AS_VALUE` property set to true cannot be nested. For example, the attribute to track PAPI counters cannot be nested, but the attribute to track the time spent inside a routine is nested.

Blackboards and Snapshots

Whenever a performance measurement is made by use of Caliper's measurement API, the values of one or more attributes are updated in an internal

data-structure referred to as the *blackboard*. This blackboard is a runtime buffer that is used to combine active attributes, and is updated by Caliper data providers (annotations).

A *snapshot* saves the current context of the blackboard. A snapshot can be triggered independently of blackboard updates. Additional information can be added to the snapshot via callbacks to snapshot events.

Services

Caliper *services* are the basic building blocks that can be combined freely to realize advanced profiling/tracing capabilities. Services are essentially plugins that register callbacks for events of interest. During Caliper initialization, the registered initialization function of each required service is invoked, and the service then performs start-up related tasks inside this initialization function.

An example of a service is the *MPI* service. The MPI service keeps track of the time spent inside MPI calls by utilizing the PMPI interface. The *recorder* service writes Caliper snapshot records into a file using a custom text-based I/O format. The recorder service in conjunction with the MPI service can be used to gather a basic profile of an MPI application. Figure 13 is an illustration of this use case.

MPI_T Service: Supporting Performance Introspection

This section describes the design of the MPI_T service that performs runtime MPI library introspection through the MPI_T interface. As of the time being, this

service does not support performance monitoring or tuning through the MPI_T interface.

Service Registration

During the registration phase for the MPI_T service, the MPI_T performance session is created. The handles for all the PVARs exported at the time of service registration are allocated. It is important to note that service registration may happen before `MPI_Init` is invoked. If this is the case, the number of PVARs exported may be zero. The design should account for this scenario.

In the next section, I shall discuss the complexities that arise with allocating handles for PVARs in detail, and how the design addresses these issues.

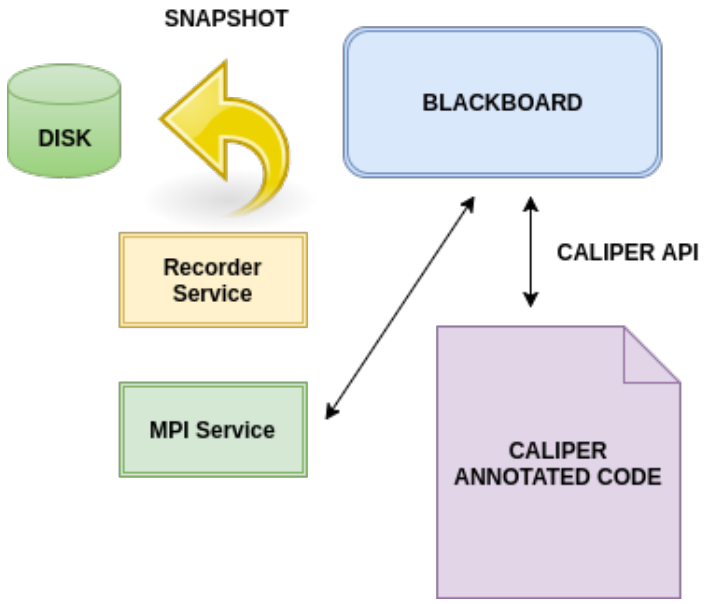

FIGURE 13. MPI profiling: Caliper service flow

PVAR Handle Allocation

Before a tool can read the value of a PVAR, it must first allocate a *handle* for the PVAR. The MPI_T interface specifies a function that allows a tool to know the number of PVARs exported by an MPI implementation at any given point in time. Recall that the number of PVARs exported can dynamically increase, and that PVARs can be bound to MPI objects. Like TAU, Caliper only supports PVARs that are exported immediately after `MPI_Init` by invoking the handle allocation routine inside the PMPI wrapper for `MPI_Init`. Any increase in the number of PVARs after this point is ignored by Caliper.

PVARs can be bound to MPI objects, and such PVARs can provide fine-grained detail about MPI. For example, a PVAR representing the number of messages sent can potentially be bound to an MPI communicator object. This way, it would be possible for a tool to distinguish the quantity of communication across communicators/process groups instead of presenting an aggregated view to the user.

Handles for PVARs not bound to any object (`MPI_T_BIND_NO_OBJECT`) can be allocated at any time — specifically, this is done inside the registration phase for the MPI_T service, and inside the PMPI wrapper for `MPI_Init`. In order to allocate handles for PVARs bound to MPI objects, we need a reference (address) to the MPI object in question. The ideal location for the MPI_T service to grab these references would be during MPI object creation. Briefly, the following steps are necessary to allocate such handles:

– Identify the corresponding MPI object creation routine for the object in question

– Intercept the object creation routine (through PMPI)

– Allocate handles for all PVARs bound to the given object type

It is possible that multiple handles are associated with a PVAR bound to an MPI object — one for each such MPI object created. The supported MPI object types in MPI_T and their corresponding object routines used to allocate handles are presented in Table 7.

TABLE 7. Caliper: PVAR handle allocation routines for supported MPI objects

MPI Object Type	MPI Object Creation Routine
MPI Communicator	MPI_Comm_Create
MPI Error Handler	MPI_Err_handler
MPI File	MPI_File_open
MPI Groups	MPI_Group_create
MPI Reduction Operators	MPI_Op_create
MPI Info Objects	MPI_Info_create
MPI Window Objects	MPI_Win_create
MPI Datatypes	*Not Supported*
MPI Message Objects	*Not Supported*
MPI Request Objects	*Not Supported*

PVAR Classes and Notion of Aggregability

Depending on what they represent, PVARs are categorized by the MPI standard into counters, state variables, watermarks, etc., and are handled differently by Caliper. We define the notion of *aggregatability* as follows: Any PVAR on which it is *meaningful* to apply one or more of the operators — SUM, MAX, MIN, AVG, COUNT is defined as aggregatable.

Along with other information, a call to `MPIT_pvar_get_info` returns the *CLASS* to which the PVAR belongs. Below we describe the various PVAR classes supported by the MPI standard and how each class is handled by Caliper:

– `MPI_T_PVAR_CLASS_TIMER`, `MPI_T_PVAR_CLASS_AGGREGATE`, `MPI_T_PVAR_CLASS_COUNTERS`: These are free-counting, monotonically increasing values. As such, they are not aggregatable, but by storing the previous value for these counters and timers, the difference between the current and previous value is a derived metric that is aggregatable by use of *SUM*, *MAX*, *MIN*, *AVG* operators. Storing this difference is more useful than just the raw counter values, as one would typically be interested in the *change* caused to any of these PVARs rather than the raw value itself.

– `MPI_T_PVAR_CLASS_STATE`: Represents MPI state at any instant in time. Non-aggregatable value.

– `MPI_T_PVAR_CLASS_SIZE`: Represents size of an MPI resource. Non-aggregatable value.

– `MPI_T_PVAR_CLASS_LEVEL`, `MPI_T_PVAR_CLASS_PERCENTAGE`: Represents the instantaneous level or percentage utilization of an MPI resource. It is meaningful to apply the *AVG*, *MIN*, *MAX* operators, and hence these classes are aggregatable.

– `MPI_T_PVAR_CLASS_HIGHWATERMARK`, `MPI_T_PVAR_CLASS_LOWWATERMARK`: As such, these classes are non-aggregatable. However, one can define aggregatable derived metrics out of these PVARs. Specifically, Caliper defines two derived metrics — A boolean that tells us if the watermark has gone up from the last time it was read, and a double value specifying the *change* in the value between successive reads. Both of these derived metrics are aggregatable quantities as one can apply the *COUNT* and/or *SUM* operator to them.

– `MPI_T_PVAR_CLASS_GENERIC`: PVARs that do not fall into any of the above classes. These PVARs would need to be handled on a case-by-case basis, and thus, for now, we define these to be non-aggregatable.

Creating Caliper Attributes for PVARs

The basic data unit in Caliper is an attribute. An attribute is a key-value pair that has certain properties. For each PVAR exposed by the MPI library, Caliper defines an attribute with the same name as the PVAR. Each PVAR attribute has the following properties:

– `CALI_ATTR_AS_VALUE` - We do not want "stacking" semantics for PVAR values. They should be treated much the same way as PAPI counters.

– `CALI_ATTR_SCOPE_PROCESS` - PVARs are defined on a per-rank basis

– `CALI_ATTR_SKIP_EVENTS` - We do not want callbacks to be triggered every time the attribute for a PVAR is updated

– `Metadata (class.aggregatable)` - Boolean value specifying if the PVAR is aggregatable or not. Aggregatability is determined based on the class to which a PVAR belongs.

Apart from creating a Caliper attribute for each PVAR exported, two additional attributes are created for each *watermark class* PVAR exported — one that represents the number of times the watermark changes, and another that represents the cumulative change in the watermark PVAR.

Sampling and Storing PVARs in Snapshots

All PVARs exported by the MPI library are queried when a snapshot is triggered. For example, when the MPI service is enabled, snapshots are taken every time an MPI call is made. By integrating the MPI_T service along with the MPI service, we would be able to determine how various MPI function calls contribute to changes in PVAR values. One can gather meaningful information by aggregating PVAR values using MPI function names or annotated code regions as keys — this would be particularly helpful in attributing MPI inefficiencies down to source code sections or MPI routines.

Depending on the class of the PVAR, we either store the raw value read from the interface in the snapshot, or a derived metric. Below, we describe various PVAR classes are represented in the snapshot:

- `MPI_T_PVAR_CLASS_TIMER`, `MPI_T_PVAR_CLASS_AGGREGATE`, `MPI_T_PVAR_CLASS_COUNTERS`: We store the difference between the current value and the previous for such PVARs in the snapshot. Storing and aggregating this derived value is more meaningful than storing the raw value — it helps us answer questions such as:
  - How do different MPI functions contribute to this PVAR?
  - Which MPI function is responsible for the highest value?
- `MPI_T_PVAR_CLASS_STATE`, `MPI_T_PVAR_CLASS_SIZE`: The raw values of such PVARs are stored in the snapshot. It may be more meaningful to view changes of these PVARs *over time*, such as in a trace.

- `MPI_T_PVAR_CLASS_HIGHWATERMARK`, `MPI_T_PVAR_CLASS_LOWWATERMARK`: Along with storing the raw value for watermark PVARs, we store the derived metrics that represent the number of times the watermark changed, along with how much the watermark changed in the snapshot. By aggregating across MPI functions for example, we can answer questions such as:
  - Which function most frequently pushed up/down a watermark?
  - Which function was responsible for the highest cumulative change in a given watermark?
- `MPI_T_PVAR_CLASS_LEVEL`, `MPI_T_PVAR_CLASS_PERCENTAGE`: The raw values of such PVARs are stored in the snapshot. It maybe meaningful to view the average, maximum, or minimum value for these PVARs aggregated across MPI functions.

Target Applications

LULESH

LULESH [32] is a mini-app developed at the Lawrence Livermore National Laboratory (LLNL). It is a typical hydrocode, and approximates the hydrodynamics equations discretely. LULESH has been ported to multiple platforms and programming models. In this study, we use the CPU-only version that uses MPI for parallelization. We use an LLNL-internal version of LULESH that has been annotated using Caliper. Specifically, all MPI functions along with some important application-level routines and loops are annotated using Caliper.

Due to restricted access to Caliper-annotated applications, our study has been limited to only one application.

Usage Scenarios

As of the writing of this document, Caliper only supports performance introspection through the MPI_T interface, so the set of usage scenarios is limited in variety. The design of the performance introspection support in Caliper was carried out with one specific goal — to analyze PVAR values *across annotated code sections* using the *aggregate* service.

In other words, we would like to *attribute* changes in PVAR values to specific code regions or function invocations. In this way, we hope that MPI_T would allow us to narrow down performance inefficiencies to specific locations in the source code. The following two usage scenarios are an attempt to showcase this design.

Detecting Performance Inefficiencies in MPI

With this use case, we would like to answer this specific question — What are the contributions of different MPI routines to PVARs that represent an *aggregatable* quantity such as memory allocated within MPI?

Sampling and storing PVARs is likely to be more effective if the context information is stored along with it for analysis later on. If we are able to identify the contributions of various MPI routines to the final sampled value of a PVAR at the end of a run, this could aid in narrowing down causes for performance inefficiencies *within* the MPI library itself.

As discussed in an earlier section introducing Caliper concepts, *services* act as basic building blocks for realizing more advanced functionality. Below, I briefly discuss the key services (apart from the MPI_T service) that are used to enable such a functionality:

Event Service

The event trigger service triggers snapshots when attributes are updated. Recall that whenever a snapshot is triggered, the MPI_T interface is queried. Through the environment variable `CALI_EVENT_TRIGGER`, the user can specify a list of attributes whose updates triggers a snapshot. Attributes that have the `CALI_ATTR_SKIP_EVENTS` property set do not trigger snapshots. PVAR attributes have this property set to true.

MPI Service

This service records MPI operations and the MPI rank. This service utilizes the PMPI interface to keep a track of the program execution time spent inside MPI. MPI function names are stored in the special attribute `mpi.function` and the MPI rank in the `mpi.rank` attribute.

Aggregate Service

The aggregate service accumulates aggregation attributes (e.g. time durations) of snapshots with a similar *key*, creating a profile. The environment variable `CALI_AGGREGATE_KEY` is used to specify the colon-separated list of attributes that are used for the aggregation key. It is important to note that these attributes can not have the `AS_VALUE` property set to true. An example of such an attribute would be `mpi.function` or `mpi.rank`.

Through the environment variable `CALI_AGGREGATE_ATTRIBUTES`, the user can specify the list of aggregation attributes. These attributes must have the `AS_VALUE` property set to true. All PVARs are candidates for attributes to form this list.

The aggregate service aggregates values of aggregation attributes from all input snapshots with similar aggregation keys.

Report Service

The report service aggregates, formats, and writes collected Caliper records into files or stdout on Caliper flush events (typically, at program end). By default, the report service prints a tabular, human-readable report of the collected snapshots.

I now describe how these services enable us to detect performance inefficiencies within MPI. The event service is used to trigger snapshots with the `mpi.function` attribute set as the `CALI_EVENT_TRIGGER`. This means that snapshots are triggered, and PVARs are sampled every time an MPI call is made. The values of these PVARs are stored in the snapshot along with value of the `mpi.function` attribute (MPI call) that triggered the snapshot. In this way, context information is stored along with the actual the values of the PVARs.

The `CALI_AGGREGATE_KEY` is a combination of the `mpi.function` and `mpi.rank` attributes. The `CALI_AGGREGATE_ATTRIBUTES` can be any combination of PVAR attribute names. Unfortunately, Caliper does not offer a good visualization tool that allows us to visualize the situation where multiple aggregate attributes are used. So we are limited to using one PVAR as the aggregation attribute. The *report* service writes out the profile files (one per MPI rank) in a form that is human-readable.

The MVAPICH2 MPI library exports a PVAR with the name `num_free_calls` that represents the number of times the `free` library routine was invoked from within MPI. This PVAR belongs to the class

aggregate.max#mpit.num_free_calls	aggregate.sum#mpit.num_free_calls	aggregate.avg#mpit.num_free_calls	aggregate.count	mpi.function
59.000000	59.000000	0.000250	235921	
16.000000	30.000000	0.009542	3144	MPI_Allreduce
0.000000	0.000000	0.000000	1	MPI_Reduce
130.000000	130.000000	130.000000	1	MPI_Finalize
0.000000	0.000000	0.000000	1	MPI_Comm_size
0.000000	0.000000	0.000000	28310	MPI_Comm_rank
0.000000	0.000000	0.000000	1	MPI_Comm_group
8.000000	8.000000	8.000000	1	MPI_Comm_create
0.000000	0.000000	0.000000	66056	MPI_Irecv
0.000000	0.000000	0.000000	62911	MPI_Isend
0.000000	0.000000	0.000000	9436	MPI_Waitall
11.000000	47186.000000	0.714333	66056	MPI_Wait
0.000000	0.000000	0.000000	1	MPI_Barrier
0.000000	0.000000	0.000000	2	MPI_Wtime

FIGURE 14. PVAR aggregated across MPI routines

`MPI_T_PVAR_CLASS_COUNTER`, and is thus a free-flowing counter whose value monotonically increases. Recall that Caliper stores the *difference* between the current and last value for such PVARs in the snapshot, and not the raw value of the counter itself. By using the `mpi.function` attribute as the key, we can aggregate this PVAR across snapshots to give us a sense of which MPI function was responsible for contributing to the value of this PVAR the most.

Figure 14 is a screenshot of the Caliper profile generated for an MPI process when enabling the MPI_T service with the other services mentioned in this section. The application being profiled is LULESH (annotated with Caliper), and the MPI library being used to run the application is MVAPICH2 version 2.3b. The profile displays various basic statistics associated with aggregating the PVAR `num_free_calls` with `mpi.function` as the key. It is evident that the `MPI_Wait` routine was responsible for a disproportionately high contribution to the value of the PVAR at the end of the run. If this behavior is common across MPI applications, an MPI library developer could look into the implementation of `MPI_Wait` for inefficiencies, as each invocation of `free` (and the associated `malloc` call) is a time-consuming operation that can degrade application performance.

Detecting Application-Level Performance Inefficiences

Consider the situation where an MPI application developer has optimized the MPI implementation to a level where further optimization yields diminishing returns. In spite of having an optimized MPI library, the application may be using a sub-optimal combination of MPI routines in order to execute a specific task. One simple example is the use of point-to-point routines to implement collectives instead of the highly optimized, library-supplied collective routines. Assuming that the user is aware of the specific PVARs to analyze in order to detect such a scenario, but is unaware of the specific code locations in the application where such an inefficiency exists, we need a way to associate PVARs to specific code locations (or routines).

With an application that has been annotated using Caliper (such as the LULESH application used in our study), this goal is easy to achieve. The set of services is identical to the previously described usage scenario with two important changes. One, we now trigger the snapshots from every annotated source function (instead of every MPI call invocation). It is assumed that an attribute with the name `function` is created for annotating the source code. Each time a function call is made, the value of this attribute is updated with the name of the function being invoked. Naturally, this attribute would have stacking semantics. Two, the aggregation key contains the attribute `function`. This way, we can see how different application-level functions contribute to the value of the PVARs.

Figure 15 is a screenshot of the Caliper profile generated for the LULESH application using this setup. The PVAR `num_free_calls` is aggregated across application-level routines. It is evident that the `CalcQForElems` and the

`CalcForceForNodes` routines were responsible for a large share of the final sampled value for this PVAR. By annotating the source code appropriately, one could use this setup to narrow down inefficiencies to specific loops or even line numbers.

Although the two usage scenarios presented above are similar with respect to the set of services used, the MPI_T sampling overheads involved vary significantly. I shall present these results in the following section.

Experiments

In this section, I present results from a study focused on determining the sampling overheads for the MPI_T support in Caliper.

Experimental Setup

All the experiments with LULESH were carried out on Quartz — a 2600-node cluster at LLNL. Each node has two Intel Xeon E5-2695 18-core processors, providing a total of 36 cores and 128 GB of main memory. All our experiments were run using 27 MPI processes on one Quartz node — a single node was sufficient for studying sampling overheads. In order to prevent performance

sum#mpit.num_free_calls	avg#mpit.num_free_calls	count	function
1.000000	1.000000	1	
0.000000	0.000000	3145	main/LagrangeLeapFrog/LagrangeNodal/CalcForceForNodes/CalcVolumeForceForElems/IntegrateStressForElems
0.000000	0.000000	6290	main/LagrangeLeapFrog/LagrangeNodal/CalcForceForNodes/CalcVolumeForceForElems/CalcHourglassControlForElems
0.000000	0.000000	3145	main/LagrangeLeapFrog/LagrangeNodal/CalcForceForNodes/CalcVolumeForceForElems/CalcHourglassControlForElems/CalcFBHourglassForceForElems
0.000000	0.000000	3145	main/LagrangeLeapFrog/LagrangeElements/UpdateVolumesForElems
0.000000	0.000000	3145	main/LagrangeLeapFrog/LagrangeNodal/CalcVelocityForNodes
0.000000	0.000000	72335	main/LagrangeLeapFrog/CalcTimeConstraintsForElems
0.000000	0.000000	34595	main/LagrangeLeapFrog/CalcTimeConstraintsForElems/CalcCourantConstraintForElems
0.000000	0.000000	15725	main/LagrangeLeapFrog/LagrangeElements
0.000000	0.000000	34595	main/LagrangeLeapFrog/CalcTimeConstraintsForElems/CalcHydroConstraintForElems
0.000000	0.000000	6290	main/LagrangeLeapFrog/LagrangeElements/CalcLagrangeElements
0.000000	0.000000	3145	main/LagrangeLeapFrog/LagrangeElements/CalcLagrangeElements/CalcKinematicsForElems
12576.000000	0.307594	40885	main/LagrangeLeapFrog/LagrangeElements/CalcQForElems
0.000000	0.000000	3145	main/LagrangeLeapFrog/LagrangeElements/CalcQForElems/CalcMonotonicQGradientsForElems
0.000000	0.000000	34595	main/LagrangeLeapFrog/LagrangeElements/CalcQForElems/CalcMonotonicQRegionForElems
0.000000	0.000000	37740	main/LagrangeLeapFrog/LagrangeElements/ApplyMaterialPropertiesForElems
0.000000	0.000000	179265	main/LagrangeLeapFrog/LagrangeElements/ApplyMaterialPropertiesForElems/EvalEOSForElems
0.000000	0.000000	440300	main/LagrangeLeapFrog/LagrangeElements/ApplyMaterialPropertiesForElems/EvalEOSForElems/CalcEnergyForElems
0.000000	0.000000	330225	main/LagrangeLeapFrog/LagrangeElements/ApplyMaterialPropertiesForElems/EvalEOSForElems/CalcEnergyForElems/CalcPressureForElems
0.000000	0.000000	34595	main/LagrangeLeapFrog/LagrangeElements/ApplyMaterialPropertiesForElems/EvalEOSForElems/CalcSoundSpeedForElems
207.000000	0.032904	6291	main
30.000000	0.009539	3145	main/TimeIncrement
0.000000	0.000000	12580	main/LagrangeLeapFrog
6621.000000	0.701749	9435	main/LagrangeLeapFrog/LagrangeNodal
34595.000000	5.500000	6290	main/LagrangeLeapFrog/LagrangeNodal/CalcForceForNodes
0.000000	0.000000	9435	main/LagrangeLeapFrog/LagrangeNodal/CalcForceForNodes/CalcVolumeForceForElems

FIGURE 15. PVAR aggregated across application routines

variation between runs, the MPI processes were pinned to cores by setting `MV2_ENABLE_AFFINITY` to true. Each experiment was performed three times, and the runtime reported here is the averaged value. Two MVAPICH2 versions were used in this study — MVAPICH2 version 2.3b and MVAPICH2 version 2.3rc2.

Results

Overhead in Enabling MPI_T

We first study how overheads vary when the number of PVARs exported by the MPI implementation differs. MVAPICH2 version 2.3b exports 73 PVARs, while MVAPICH2 version 2.3rc2 exports 402 PVARs. For this experiment, snapshots were triggered (and MPI_T subsequently sampled for *all* PVARs) whenever an MPI call was made. We define the following terms:

- **Baseline** — Caliper-annotated LULESH profiled without any services enabled
- **Without MPI_T** — Caliper-annotated LULESH profiled with the MPI, report, timestamp, aggregate, and event services enabled
- **With MPI_T** — Caliper-annotated LULESH profiled with the MPI, MPI_T, report, timestamp, aggregate, and event services enabled

Figure 16 depicts an expected outcome — the overheads are directly proportional to the number of PVARs exported. For MVAPICH2 version 2.3b, this corresponds to overheads of 76% over the baseline, whereas for MVAPICH2 version 2.3rc2, overheads are 143% over the corresponding baseline. None of these numbers are in an acceptable range — this suggests that it is too expensive to sample *all* PVARs, every time an MPI call is made.

As alluded to in the section on usage scenarios, the snapshot-triggering mechanism has a significant impact on the MPI_T sampling overheads. Specifically, we consider two situations:

– Snapshots triggered when an MPI call is made

– Snapshots triggered when a Caliper-annotated application routine is invoked

For this experiment, the MVAPICH2 version 2.3rc2 was used. Figure 17 suggests that the overheads involved in triggering a snapshot and sampling MPI_T from application-level routines are even higher than the situation where snapshots are triggered every time an MPI call is made. Specifically, we see overheads of 207% when triggering snapshots from application-level routines, and overheads of 143% when triggering snapshots from MPI routines.

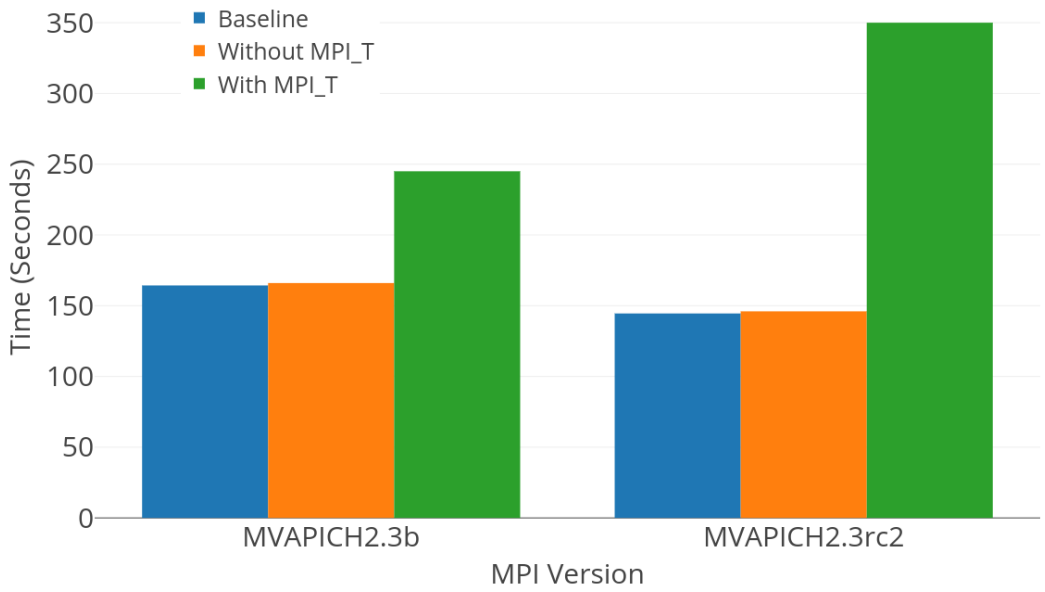

FIGURE 16. Effect of number of PVARs exported on MPI_T overhead

This is a sobering result — even when the level of application instrumentation is low-to-moderate (such as in LULESH), MPI_T sampling overheads are not acceptable. This would likely be the case in other scientific applications that are iterative (or depend on a timestep loop). A compromise solution in this situation may be to sample only a subset of PVARs — perhaps by providing the user an environment variable to specify such a subset. But such a solution assumes that the user is already aware of the exact names (or indices) of the PVARs — this is a poor assumption to make in the context of MPI_T.

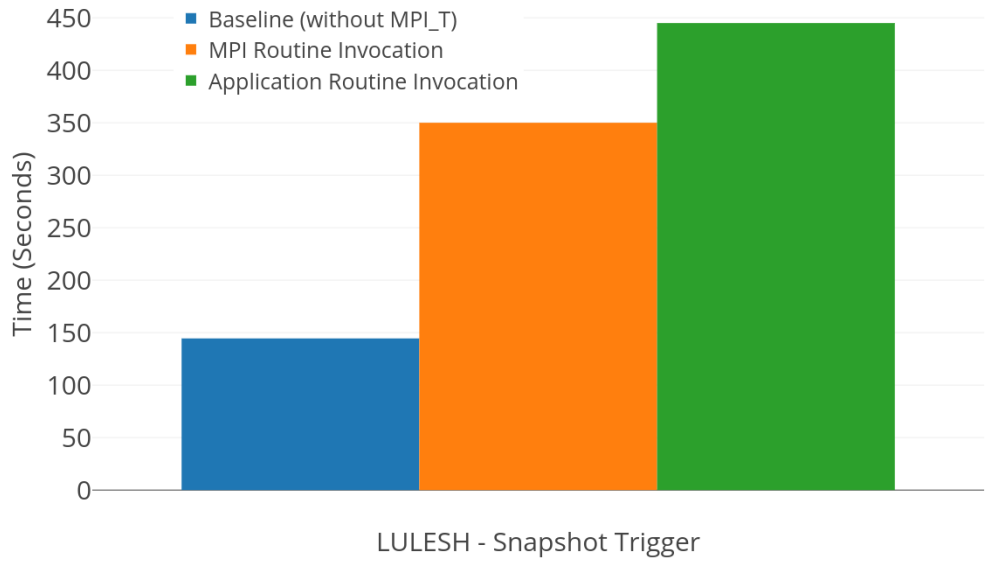

FIGURE 17. Effect of snapshot trigger mechanism on MPI_T overhead

Implementation Challenges and Issues

Crashes with OpenMPI

The MPI_T support in Caliper was designed to be specifically used with the OpenMPI implementation. When this research was being carried out, OpenMPI was the only MPI library that had support for PVARs bound to MPI objects. However, we noted that the application crashed when we were trying to allocate handles for PVARs bound to MPI objects. This issue was communicated to the developers of the OpenMPI library. Unfortunately, this issue was not resolved in time, and we had to use MVAPICH2 for testing purposes.

Lack of GUI support in Caliper

When this research was being conducted, Caliper did not have GUI support for visualizing profiles, nor did it have support for writing out traces in a standard format. Caliper had basic text-based support for viewing and analyzing the collected performance data. Given the relatively large number of PVARs exported by MVAPICH2 (order of a 100), the basic text-based support was not sufficient to display all the data. We had to resort to viewing only a subset of PVARs at any given time.

Summary

This chapter has described the design and implementation of the MPI_T support in Caliper. Through experiments, we have demonstrated the capabilities and limitations of the performance introspection support in Caliper. In the next

chapter, we shall present a discussion focusing on the design differences between the MPI_T support in TAU and Caliper.

CHAPTER V

DISCUSSION

MPI_T allows a performance profiler such as TAU or Caliper to play a more active role in MPI performance engineering. As I have demonstrated with experiments with TAU on AmberMD, SNAP, and 3DStencil, there can be significant memory savings in tracking and freeing unused virtual buffers inside MVAPICH2. Such opportunities for fine-tuning MPI library behavior would not have been possible without a close interaction between this two software.

Design Differences Between TAU and Caliper

Although both TAU and Caliper offer MPI performance introspection capabilities through MPI_T, they differ significantly in the design and implementation of this support. TAU primarily relies on an interrupt-based mechanism to sample the MPI_T interface, while Caliper relies on an *event* to trigger the MPI_T sampling routine. This has far-reaching consequences to the overheads involved in introspecting the MPI_T interface.

TAU's interrupt-based mechanism is extremely light-weight — it adds almost no noticeable overhead to application runtime even when sampling at the rate of once every second. The event-based scheme implemented in Caliper is expensive — the overheads generated by both the snapshot triggering mechanisms discussed in this document are prohibitively high.

Although TAU has support for context events, PVARs are stored as user events (they are treated as regular counters) — as a result, it is not possible in the current design to add metadata information when sampling PVARs. Caliper,

on the other hand, has a more flexible API — the user can define attributes with a set of properties. Specifically, the snapshotting mechanism in Caliper offers a convenient way to add rich context to PVAR data that is collected. This enables a more *meaningful* analysis of PVAR values at the end of a profiling run — a user can attribute PVAR contributions to specific code sections.

TAU clearly has a broader level of support for MPI_T— it supports performance monitoring, autotuning, and recommendation generation through MPI_T in addition to performance introspection. The plugin design in TAU specifically enables support for a broader range of MPI implementations. Caliper at the moment does not support any of these additional features.

The lack of a GUI for performance analysis in Caliper makes in particularly hard to analyze PVARs effectively. Moreover, the profile and trace files are written in a custom format — the use of well-established tools to view these files is therefore limited. However, it must be noted that Caliper has support for a tool API that enables other tools such as TAU to "plug-in" to Caliper at runtime to extract the collected performance data. This support has not been explored in this work.

A Note on the MPI_T Interface Specification

Both these tools rely on a PMPI wrapper to generate MPI profiling information. In addition to performing this task, the wrapper in Caliper allocates handles for PVARs that are bound to MPI objects. Although not currently implemented, it is certainly feasible to implement such a functionality within TAU as well. This feature of the MPI_T specification is particularly interesting as it enables a more fine-grained performance analysis of MPI. However, support for

this feature is limited — OpenMPI is the only implementation that supports this feature.

The MPI_T standard specification allows an MPI library to dynamically export additional PVARs as and when they become available (through dynamic loading of shared objects). In my opinion, this feature can be supported by tools only by incurring a significant cost in terms of implementation complexity and performance degradation. The solution that would ensue would invariably be ugly by design. As it involves memory allocation, a tool must be careful about when it allocates handles for the additional PVARs (CVARs). Based on the experience with TAU, this certainly cannot be done inside an interrupt handler. Restricting MPI implementations in a way that ensures that they export *all* PVARs (CVARs) during `MPI_Init` can significantly reduce this complexity.

Summary

In this chapter, we have described how TAU and Caliper differ in their approach to implementing MPI_T based performance introspection. We have also discussed how these design choices affect runtime overheads. In the concluding chapter, we will describe current efforts in advancing the MPI_T support in TAU and touch upon future directions for research.

CHAPTER VI

CONCLUSION AND FUTURE WORK

This thesis presented an infrastructure dedicated to MPI Performance Engineering, enabling introspection of MPI runtimes. To serve that purpose, this infrastructure utilized the MPI Tools Information Interface, introduced in the MPI 3.0 standard.

I discussed how the TAU Performance System and MVAPICH2 could be extended to fully exploit features offered by MPI_T. I demonstrated different usage scenarios based on specific sets of MPI_T Performance and Control Variables exported by MVAPICH2. The results produced by our experiments on a combination of synthetic and production applications validate our approach and open broad perspectives for future research.

With Caliper, I presented an infrastructure that enables performance introspection through MPI_T. As compared to TAU, I chose to experiment with a different strategy to sample PVARs. I demonstrated how this strategy could lead to a more meaningful way to analyze PVARs, even if the overheads associated with this strategy were high.

Turning to the future, it is interesting to consider that MPI_T provides an opportunity to perform autotuning on an *extremely* fine-grained level — right down to message-level granularity. MVAPICH2 exports an environment variable called `MV2_RNDV_PROTOCOL` that determines the *rendezvous* protocol used on RDMA capable systems — `RDMA_WRITE` or `RDMA_READ`. Our experiments with microbenchmarks suggest that using the right RDMA protocol for a

communicating pair of processes at a given callsite can significantly improve *rendezvous* nonblocking point-to-point performance.

Detecting the runtime ordering of the posting of the nonblocking send, receive, and the corresponding wait operation is critical in determining the right RDMA protocol to use for *rendezvous* communication. The *rendezvous* protocol typically involves the exchange of control messages between the sender and receiver. MPI_T can potentially help in detecting this exchange of control messages and in ultimately determining how to tune the protocol at a fine-grained level.

We must first verify that tuning the *rendezvous* protocol at a fine-grained level can indeed lead to performance benefits. However, MVAPICH2 does not have support for tuning the `MV2_RNDV_PROTOCOL` at a fine-grained level. The value of this variable is common across MPI processes and MPI callsites and cannot be changed at runtime. So, we resort to simulating the *rendezvous* protocol using a trace-replay tool such as TraceR [33] in order to confirm or reject this hypothesis. We hope to show that fine-grained tuning of the *rendezvous* protocol does indeed lead to performance benefits by post-processing application traces. Through simulation, we also hope to demonstrate a mechanism to profile and tune the *rendezvous* protocol dynamically at runtime.

We plan to enrich our infrastructure by also exploring the following areas of research:

- Develop an infrastructure to express autotuning policies in a more generic fashion
- Enrich MPI_T support in MVAPICH2 to enable introspection and tuning for a wide range of applications and communication patterns

– Study the challenges in providing an interactive performance engineering functionality for end users

With respect to future Caliper research, it currently lacks a mechanism to tune the MPI library at runtime using the MPI_T interface. We plan to add support for MPI_T based autotuning in Caliper. A prospective research idea is to integrate Caliper with TAU through Caliper's tool API. This way, Caliper can leverage TAU's MPI_T infrastructure for performance autotuning or recommendations. Caliper also lacks a proper tool for performance analysis — through this integration, performance data collected through Caliper can be analyzed using TAU's rich support for profiling and tracing tools.

REFERENCES CITED

[1] Srinivasan Ramesh, Aurèle Mahéo, Sameer Shende, Allen D. Malony, Hari Subramoni, and Dhabaleswar K. Panda. MPI performance engineering with the MPI tool interface: the integration of MVAPICH and TAU. In *Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI/USA 2017, Chicago, IL, USA, September 25-28, 2017*, pages 16:1–16:11, 2017. doi: 10.1145/3127024.3127036.

[2] Srinivasan Ramesh, Aurèle Mahéo, Sameer Shende, Allen D. Malony, Hari Subramoni, Amit Ruhela, and Dhabaleswar K. (DK) Panda. MPI Performance Engineering with the MPI Tool Interface: the Integration of MVAPICH and TAU. *Parallel Computing*, 2018. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2018.05.003.

[3] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.1, June 4th 2015. `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf` (June. 2015).

[4] Edward Karrels and Ewing Lusk. Performance analysis of mpi programs. *Environments and Tools for Parallel Scientific Computing*, pages 195–200, 1994.

[5] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006. ISSN 1094-3420. doi: 10.1177/1094342006064482. `http://tau.uoregon.edu`.

[6] Jiuxing Liu, Jiesheng Wu, Sushmitha P Kini, Pete Wyckoff, and Dhabaleswar K Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 295–304. ACM, 2003. `http://mvapich.cse.ohio-state.edu/`.

[7] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 97–104. Springer, 2004.

[8] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[9] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par 08, page 7888, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85450-0. doi: 10.1007/978-3-540-85451-7_9.

[10] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: performance introspection for HPC software stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 47. IEEE Press, 2016. `https://github.com/LLNL/Caliper`.

[11] Swann Perarnau, Rinku Gupta, Pete Beckman, et al. Argo: An Exascale Operating System and Runtime, 2015. `http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/poster_files/post298s2-file2.pdf`.

[12] Swann Perarnau, Rajeev Thakur, Kamil Iskra, Ken Raffenetti, Franck Cappello, Rinku Gupta, Pete Beckman, Marc Snir, Henry Hoffmann, Martin Schulz, and Barry Rountree. Distributed Monitoring and Management of Exascale Systems in the Argo Project. In *Proceedings of the 15th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 9038*, pages 173–178, New York, NY, USA, 2015. Springer-Verlag New York, Inc. ISBN 978-3-319-19128-7. doi: 10.1007/978-3-319-19129-4_14.

[13] Rainer Keller, George Bosilca, Graham Fagg, Michael Resch, and Jack J. Dongarra. Implementation and Usage of the PERUSE-Interface in Open MPI. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany, September 2006. Springer-Verlag.

[14] Tanzima Islam, Kathryn Mohror, and Martin Schulz. Exploring the Capabilities of the New MPI_T Interface. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 91:91–91:96, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2875-3. doi: 10.1145/2642769.2642781. `https://computation.llnl.gov/projects/mpi_t/gyan`.

[15] Esthela Gallardo, Jerome Vienne, Leonardo Fialho, Patricia Teller, and James Browne. MPI Advisor: A Minimal Overhead Tool for MPI Library Performance Tuning. In *Proceedings of the 22Nd European MPI Users' Group Meeting*, EuroMPI '15, pages 6:1–6:10, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3795-3. doi: 10.1145/2802658.2802667.

[16] Esthela Gallardo, Jrme Vienne, Leonardo Fialho, Patricia Teller, and James Browne. Employing MPI_T in MPI Advisor to optimize application performance. *The International Journal of High Performance Computing Applications*, 0(0):1094342016684005, 0. doi: 10.1177/1094342016684005.

[17] Jeffrey Vetter and Chris Chambreau. mpiP: Lightweight, scalable mpi profiling. 2005. `http://mpip.sourceforge.net`.

[18] Mohamad Chaarawi, Jeffrey M. Squyres, Edgar Gabriel, and Saber Feki. *A Tool for Optimizing Runtime Parameters of Open MPI*, pages 210–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-87475-1. doi: 10.1007/978-3-540-87475-1_30.
`https://www.open-mpi.org/projects/otpo/`.

[19] M. Gerndt and M. Ott. Automatic Performance Analysis with Periscope. *Concurr. Comput. : Pract. Exper.*, 22(6):736–748, April 2010. ISSN 1532-0626. doi: 10.1002/cpe.v22:6. `http://periscope.in.tum.de/`.

[20] Anna Sikora, Eduardo César, Isaías Comprés, and Michael Gerndt. Autotuning of MPI Applications Using PTF. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, SEM4HPC '16, pages 31–38, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4351-0. doi: 10.1145/2916026.2916028.

[21] Simone Pellegrini, Thomas Fahringer, Herbert Jordan, and Hans Moritsch. Automatic Tuning of MPI Runtime Parameter Settings by Using Machine Learning. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 115–116, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0044-5. doi: 10.1145/1787275.1787310.

[22] Kevin Huck, Sameer Shende, Allen Malony, Hartmut Kaiser, Allan Porterfield, Rob Fowler, and Ron Brightwell. An Early Prototype of an Autonomic Performance Environment for Exascale. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '13, pages 8:1–8:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2146-4. doi: 10.1145/2491661.2481434.
`http://khuck.github.io/xpress-apex/`.

[23] Kathleen A Lindlan, Janice Cuny, Allen D Malony, Sameer Shende, Bernd Mohr, Reid Rivenburgh, and Craig Rasmussen. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 49–49. IEEE, 2000.

[24] David A Case, Thomas E Cheatham, Tom Darden, Holger Gohlke, Ray Luo, Kenneth M Merz, Alexey Onufriev, Carlos Simmerling, Bing Wang, and Robert J Woods. The Amber biomolecular simulation programs. *Journal of computational chemistry*, 26(16):1668–1688, 2005. `http://ambermd.org/`.

[25] Robert J Zerr and Randal S Baker. SNAP: SN (discrete ordinates) application proxy: Description. *Los Alamos National Laboratories, Tech. Rep. LAUR-13-21070*, 2013. `https://github.com/lanl/SNAP/`.

[26] Ray E Alcouffe, Randal S Baker, Jon A Dahl, Scott A Turner, and Robert Ward. PARTISN: A time-dependent, parallel neutral particle transport code system. *Los Alamos National Laboratory, LA-UR-05-3925 (May 2005)*, 2005.

[27] Lawrence Livermore. Los Alamos, and Sandia National Laboratories. The accelerated strategic computing initiative (ASCI) sweep3d benchmark code, 1995.

[28] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009. `https://mantevo.org/`.

[29] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol (SHArP): a hardware architecture for efficient data reduction. In *Proceedings of the First Workshop on Optimization of Communication in HPC*, pages 1–10. IEEE Press, 2016.

[30] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008. `www.vampir.eu`.

[31] TACC Stampede cluster. The University of Texas at Austin: `http://www.tacc.utexas.edu`.

[32] Ian Karlin, Jeff Keasler, and JR Neely. Lulesh 2.0 updates and changes. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013.

[33] Bilge Acun, Nikhil Jain, Abhinav Bhatele, Misbah Mubarak, Christopher D Carothers, and Laxmikant V Kale. Preliminary evaluation of a parallel trace replay tool for hpc network simulations. In *European Conference on Parallel Processing*, pages 417–429. Springer, 2015.
`https://github.com/LLNL/TraceR`.