Parallel Computing 77 (2018) 19–37

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

MPI performance engineering with the MPI tool interface: The integration of MVAPICH and TAU

Srinivasan Ramesh[a,*], Aurèle Mahéo[a], Sameer Shende[a], Allen D. Malony[a], Hari Subramoni[b], Amit Ruhela[b], Dhabaleswar K. (DK) Panda[b]

[a] Department of Computer and Information Science, University of Oregon, 1477 E 13th Ave, Eugene, Oregon 97403, USA
[b] Department of Computer Science and Engineering, The Ohio State University, 2015 Neil Ave, Columbus, Ohio 43210, USA

ARTICLE INFO

Article history:
Received 1 December 2017
Revised 27 February 2018
Accepted 17 May 2018
Available online 18 May 2018

Keywords:
MPI_T
Runtime introspection
Autotuning
Performance engineering
Performance recommendations
TAU
MVAPICH2
BEACON

ABSTRACT

The desire for high performance on scalable parallel systems is increasing the complexity and tunability of MPI implementations. The MPI Tools Information Interface (MPI_T) introduced as part of 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. In this paper, we propose an infrastructure that extends existing components — TAU, MVAPICH2, and BEACON to take advantage of the MPI_T interface and offer runtime introspection, online monitoring, recommendation generation, and autotuning capabilities. We validate our design by developing optimizations for a combination of production and synthetic applications. Using our infrastructure, we implement an autotuning policy for AmberMD (a molecular dynamics package) that monitors and reduces the internal memory footprint of the MVAPICH2 MPI library without affecting performance. For applications such as MiniAMR whose collective communication is latency sensitive, our infrastructure is able to generate recommendations to enable hardware offloading of collectives supported by MVAPICH2. By implementing this recommendation, the MPI time for MiniAMR at 224 processes reduces by 15%.

1. Introduction

The Message Passing Interface [1] 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 [2] interface, 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®; [3]is a popular tool that offers the user a comprehensive list of features to

* Corresponding author.
E-mail addresses: sramesh@cs.uoregon.edu (S. Ramesh), amaheo@uoregon.edu (A. Mahéo), sameer@cs.uoregon.edu (S. Shende), malony@cs.uoregon.edu (A.D. Malony), subramoni.1@osu.edu (H. Subramoni), ruhela.2@osu.edu (A. Ruhela), panda@cse.ohio-state.edu (D.K.. Panda).

https://doi.org/10.1016/j.parco.2018.05.003
0167-8191/

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 [4–7] 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 paper 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. We implement such an infrastructure with the integration of MVAPICH2 [4] and TAU [3]. We use a combination of production (AmberMD [8]), and synthetic applications (MiniAMR [9] and 3DStencil) as case studies to demonstrate the effectiveness of our design.

This paper makes the following contributions:

- Enhance MPI_T support in MVAPICH2 by developing a richer class of MPI_T performance and control variables;
- Enable performance introspection and online monitoring through tight integration between MVAPICH2, TAU, and BEACON;
- Perform runtime autotuning through MPI_T by developing plugin extensions for TAU; and
- Generate performance recommendations through plugin extensions for TAU.

The rest of the paper is organized as follows. In Section 2, we describe the MPI Tools Information Interface. In Section 3, we describe the related work in the area of runtime introspection of MPI libraries. In Section 4, we provide a background on the software components and the target applications in our study. In Section 5, we describe our contribution — a software infrastructure that extends existing software components to enable runtime introspection and tuning of MPI libraries through the MPI_T interface. In Section 6, we describe sample usage scenarios for our infrastructure. In Section 7, we present experimental results of target applications that have benefited from runtime introspection using our infrastructure. In Section 8, we discuss the impact of our research, and in Section 9, we present our conclusion and future directions for our work.

2. 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 [1]. 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.

3. 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.

3.1. Interfaces for runtime introspection

Throughout MPI's 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 [10] 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 [11], 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.

3.2. Performance recommendations

Other contributions, focusing on tuning MPI configuration parameters, provide performance recommendations to the users. MPI Advisor [12,13] 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 [14] 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) [15], 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.

3.3. 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 [16,17], 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 [18] 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.

3.4. Policy engine for performance tuning

Outside the scope of MPI, Huck et al. describe APEX [19], the *Autonomic Performance Environment for eXascale*. APEX ships with TAU, and is part of the XPRESS project, which includes a parallel programming model named OpenX, and a runtime implementing this model, HPX. Working on top of 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.

4. Background

Our work targets the development of an integrated software infrastructure that enables the use of MPI_T for performance introspection and online tuning. Here we describe the functionality of the key components and present the applications we used in our study.

4.1. TAU Performance System®;

TAU [3] is a comprehensive performance analysis toolkit that offers capabilities to instrument and measure scalable parallel applications on a variety of HPC architecture. TAU 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).

4.2. MVAPICH2

MVAPICH2 [4] 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.

4.3. BEACON

BEACON (Backplane for Event and Control Notification) [20] is a communication infrastructure, originally part of the Argo project [21]. 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, 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.

4.4. Target applications

4.4.1. AmberMD

AmberMD [8] 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.

4.4.2. SNAP

SNAP [22] is a proxy application from the Los Alamos National Laboratory that is designed to mimic the performance characteristics of PARTISN [23]. 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 [24] 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.

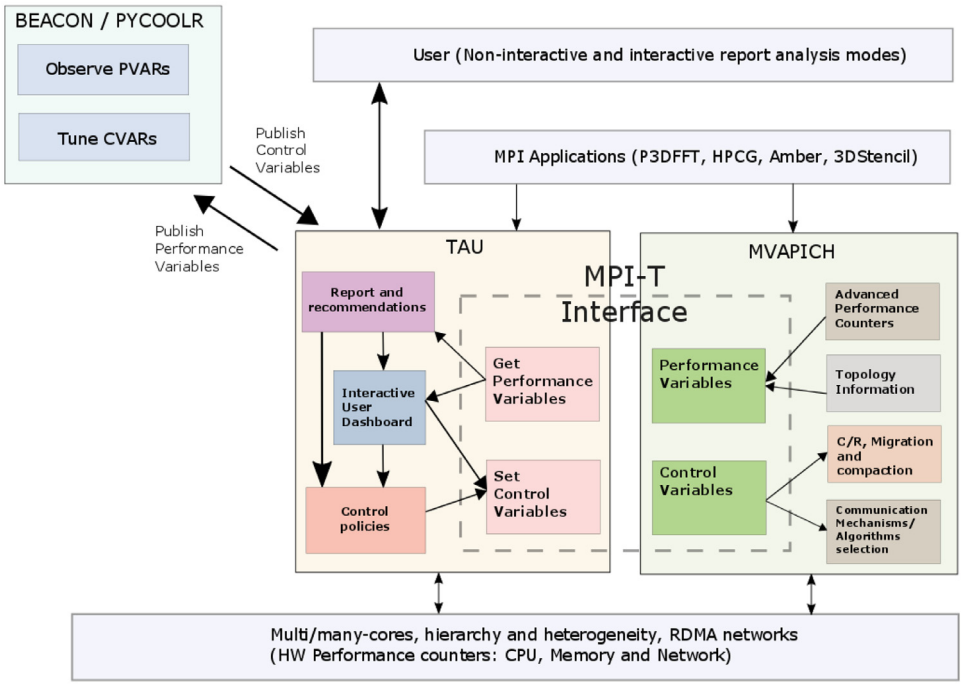


Fig. 1. Integrated MVAPICH2 and TAU infrastructure based on MPI_T.

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.

4.4.3. 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.

4.4.4. MiniAMR

MiniAMR is a mini-app that is a part of the Mantevo [9] 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.

5. Design

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. Below, we describe the design approach for MVAPICH2 and TAU integration to enable runtime introspection, performance tuning, and recommendation generation. Fig. 1 depicts the infrastructure architecture and component interactions.

5.1. 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:

5.1.1. 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.

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

5.2. 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.

5.2.1. 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 `SIGUSR` 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, etc. 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.

5.2.2. Online monitoring

Runtime introspection naturally extends to online monitoring where certain performance variables are made viewable during execution. Fig. 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.

5.2.3. 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*.

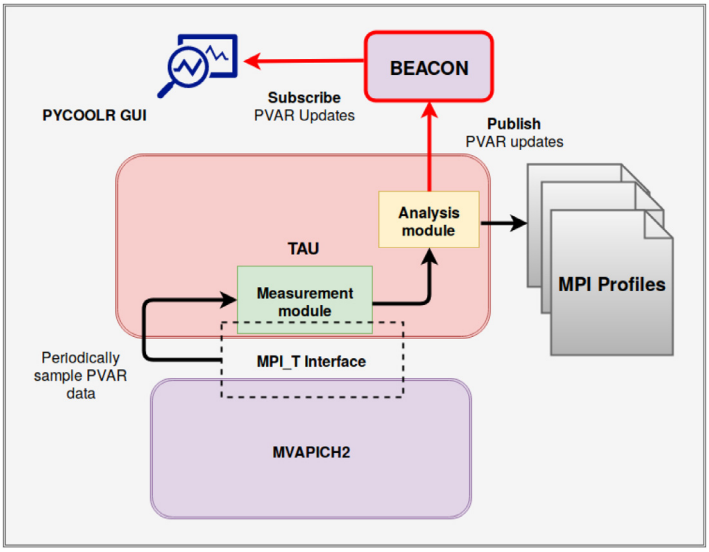


Fig. 2. Online monitoring with BEACON/PYCOOLR.

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.

As described in Section 5.2.1, 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.

5.3. 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.

5.3.1. 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 Fig. 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 — Fig. 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.

5.4. 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

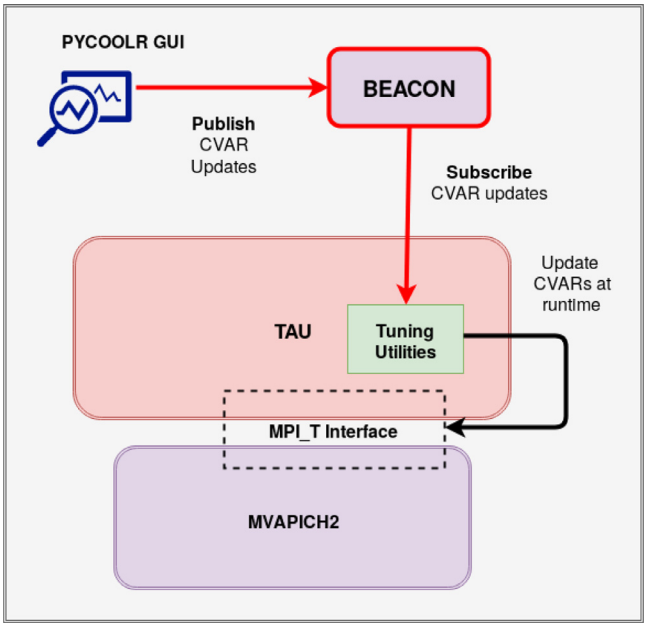


Fig. 3. User-Guided Tuning with BEACON/PYCOOLR.

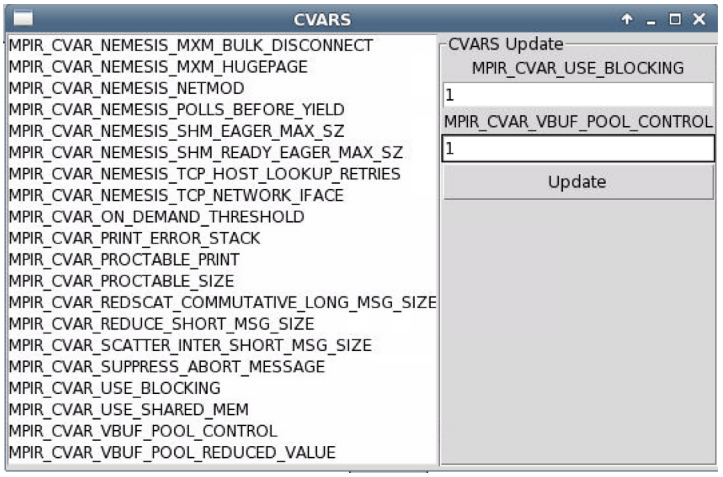


Fig. 4. Screenshot of PYCOOLR window to update CVARs.

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 *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[1] supports this plugin infrastructure.

5.4.1. Design overview

In the current design, plugins are C\C++ modules that are built into separate shared libraries. The user can specify the path to the directory containing the plugins using the environment variable `TAU_PLUGINS_PATH`. The user can also specify the plugins to be loaded using the environment variable `TAU_PLUGINS` separated by a delimiter.

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

- **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

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

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 Section 5.4.2. 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.

5.4.2. 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:

- 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.
- 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.
- 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.
- INTERRUPT_TRIGGER: TAU's sampling subsystem relies on installing an interrupt handler for the SIGUSR 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).
- END_OF_EXECUTION: When TAU has finished creating and writing the profile files for the application, this plugin event is triggered.

We plan to add to the list of supported events in future releases.

5.4.3. 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) [25].

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 FUNCTION_REGISTRATION plugin event. Recall that this event is triggered once for every function instrumented by TAU. Within the callback for the 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.

5.5. Plugins for autotuning

Fig. 5 depicts TAU plugins in the context of our MPI_T infrastructure. As discussed in Section 5.2, TAU samples PVAR data from the MPI_T interface inside a signal handler for the SIGUSR 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 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

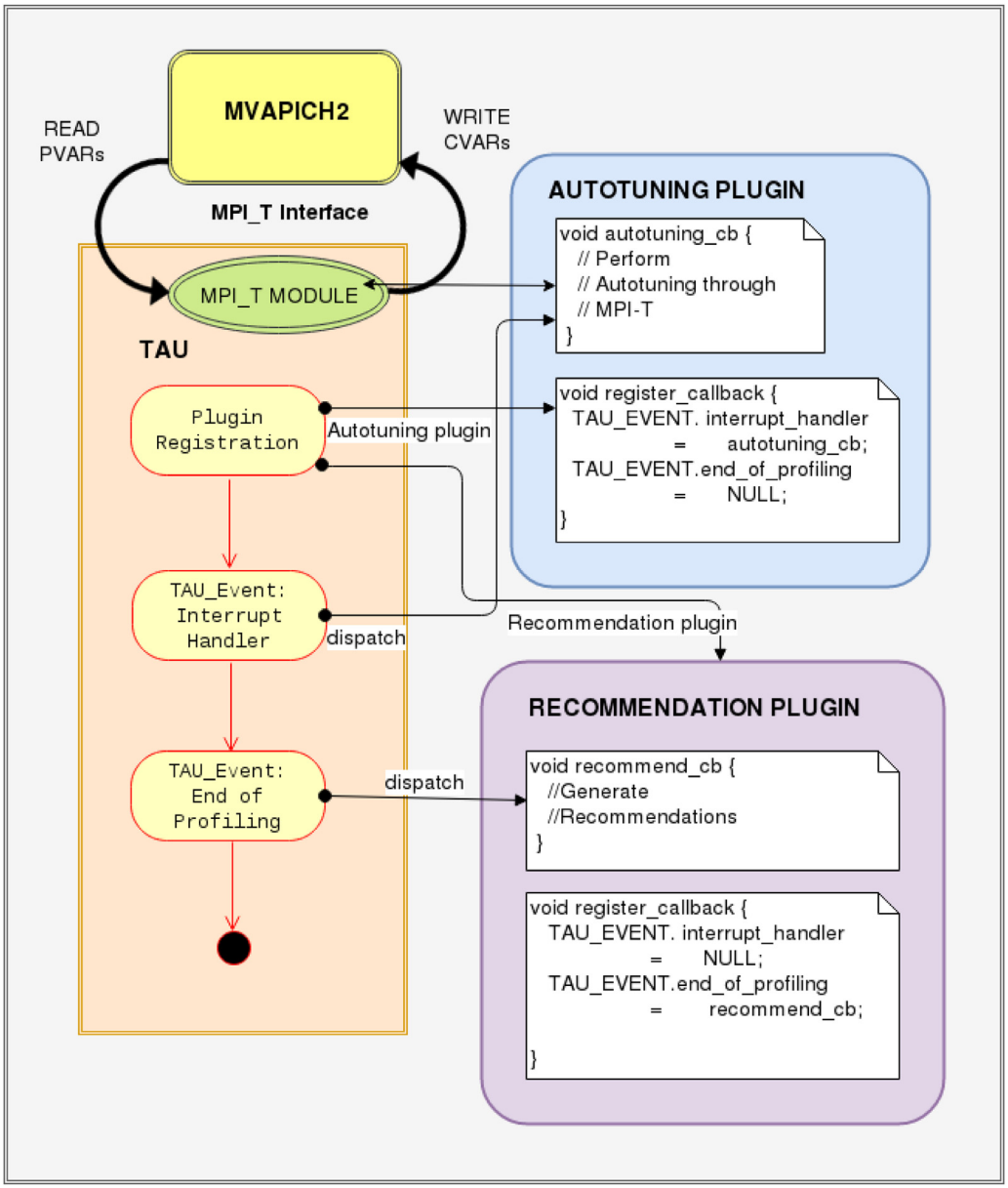


Fig. 5. Plugin Infrastructure.

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.

5.6. 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 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

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.

6. 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.

6.1. Recommendation use case: hardware offloading of collectives

MVAPICH2 now supports offloading of MPI_Allreduce to network hardware using the SHArP [26] 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.

6.2. 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.

6.2.1. 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. Fig. 6 is an enlarged Vampir [27] summary process timeline view for one iteration of the 3DStencil application before applying the Eager optimization. Fig. 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

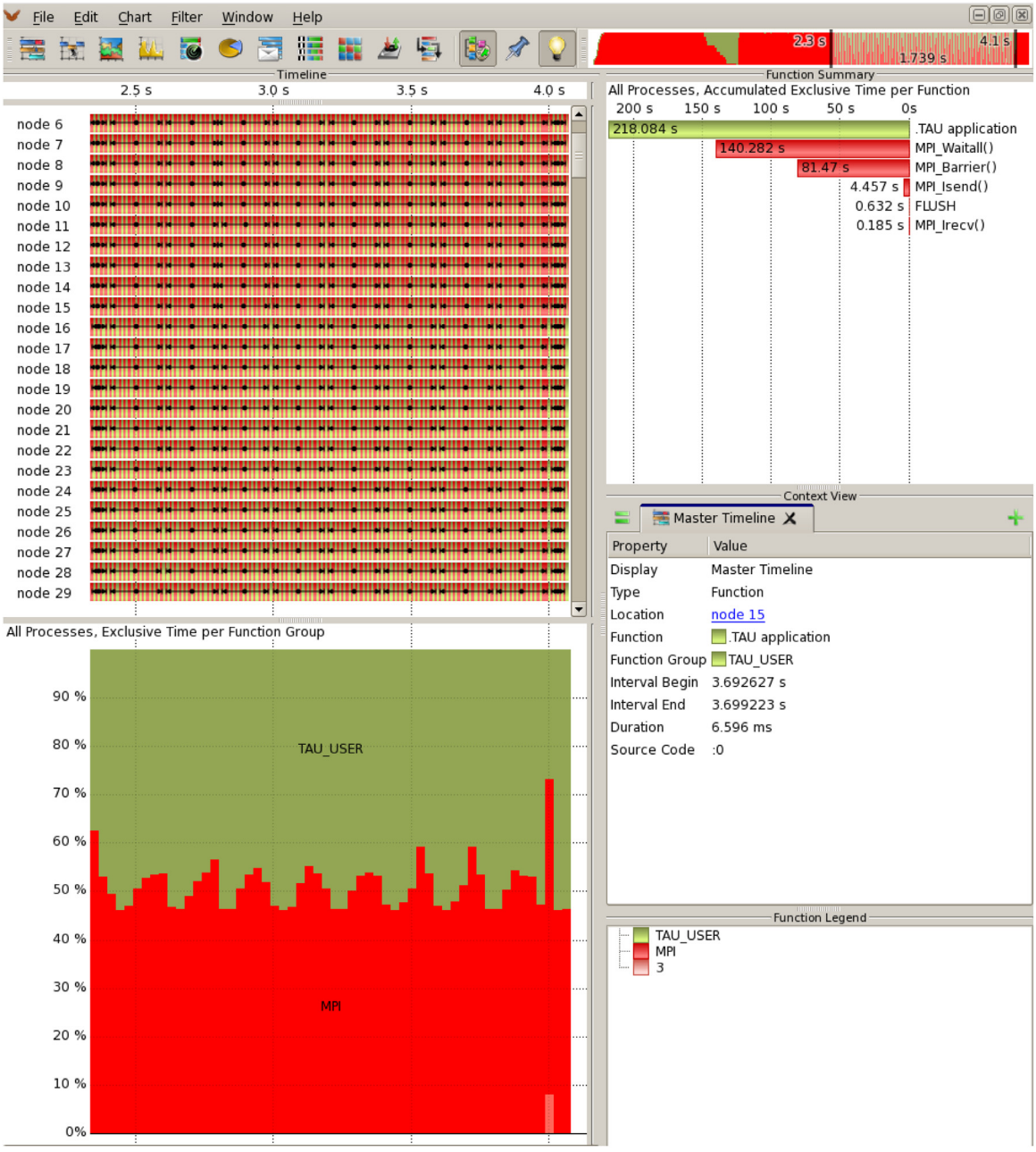


Fig. 6. Vampir [27] summary process timeline view of 3DStencil before Eager threshold tuning.

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, annotated by TAU_USER) and MPI code (red, annotated by MPI) 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.

Fig. 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 communication in the background while the sending process is busy performing the computation. The extreme right edges of Fig. 7 are to be ignored as they represent the phase where the application is performing pure communication (refer to Section 4.4.2).

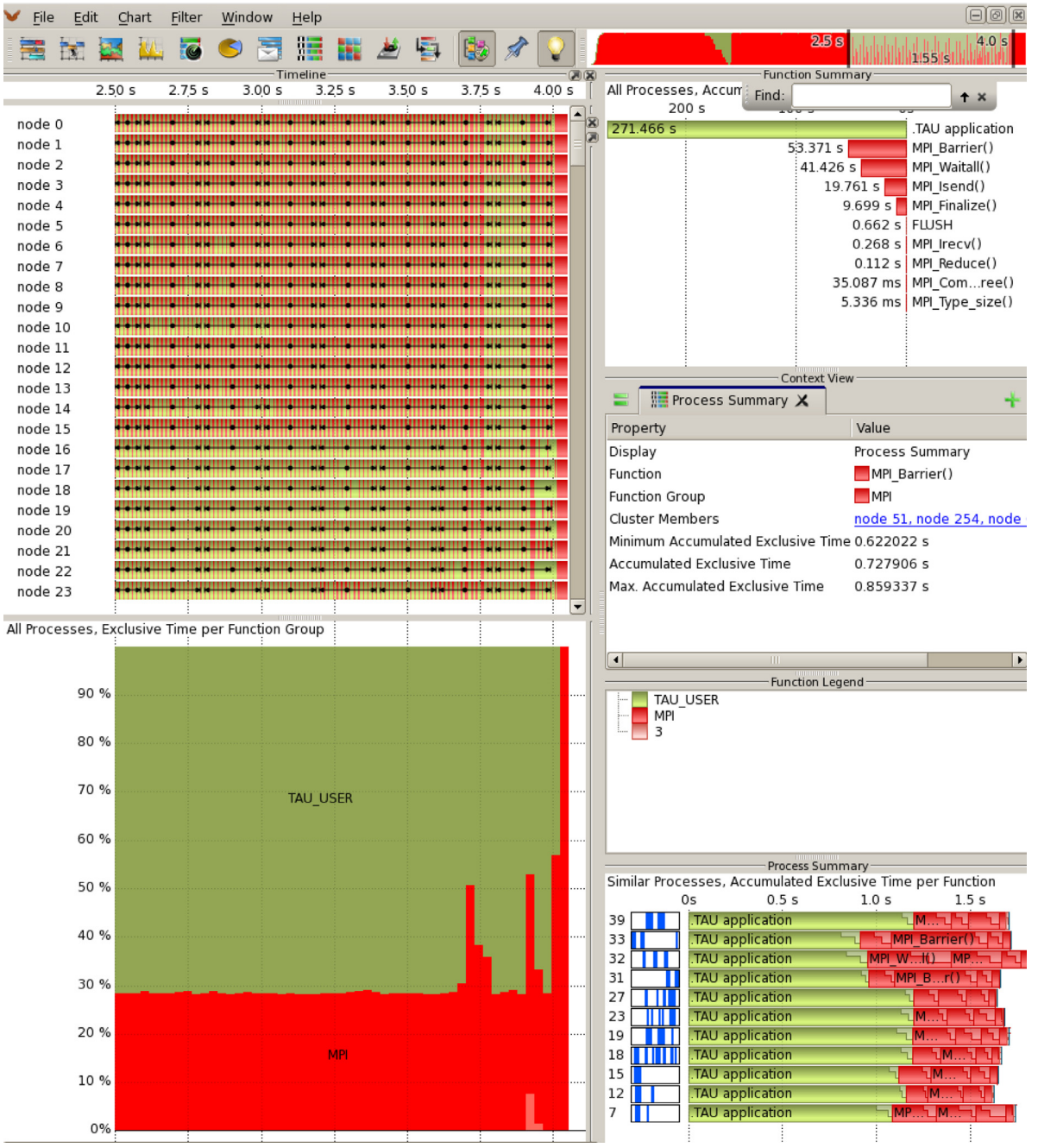
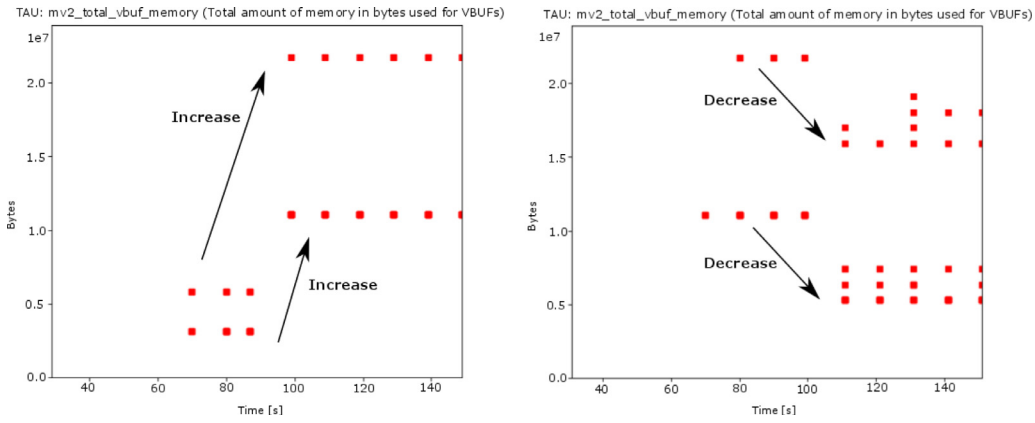


Fig. 7. Vampir [27] summary process timeline view of 3DStencil after Eager threshold tuning: Increased time spent in user code.

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) Increase in total VBUF memory when increasing Eager threshold

(b) Decrease in total VBUF memory when freeing unused VBUFs

Fig. 8. PYCOOLR screenshots depicting evolution of VBUF memory with Eager protocol.

A combination of these two factors may lead to an increase in the total VBUF memory usage inside MVAPICH2. Fig. 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 Fig. 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`. Fig. 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.

Fig. 8 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.

Table 1
AmberMD - Impact of Eager threshold and autotuning on execution time and memory usage.

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

7. Experiments

In this section, we present the results obtained from applying the autotuning and recommendation policies described in Section 6, to our target applications — AmberMD, SNAP, 3DStencil, and MiniAMR. We describe results from a study of overheads involved in enabling MPI_T in MVAPICH2 and TAU.

7.1. Experimental setup

Our experiments with AmberMD, SNAP, and 3DStencil were performed on Stampede, a 6400 node Infiniband cluster at the Texas Advanced Computing Center [28]. 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.

7.2. Results

7.2.1. 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.

7.2.2. 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:

- 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

Table 2
SNAP - Aggregate time inside various MPI functions (top five by total time).

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 (top five by message count).

MPI routine name	Callsite ID	Count	Average message size (Bytes)
MPI_Isend	1	114,348,672	18,300
MPI_Allreduce	7	25,600	1200
MPI_Send	18	2400	1920
MPI_Bcast	11	1024	120
MPI_Bcast	15	1024	120

Table 4
SNAP - Impact of Eager threshold and autotuning on execution time and memory usage.

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

Table 5
3DStencil - Impact of Eager threshold and autotuning on execution time and memory usage.

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

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 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.

7.2.3. 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 6
MiniAMR - Impact of hardware offloading on application runtime.

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

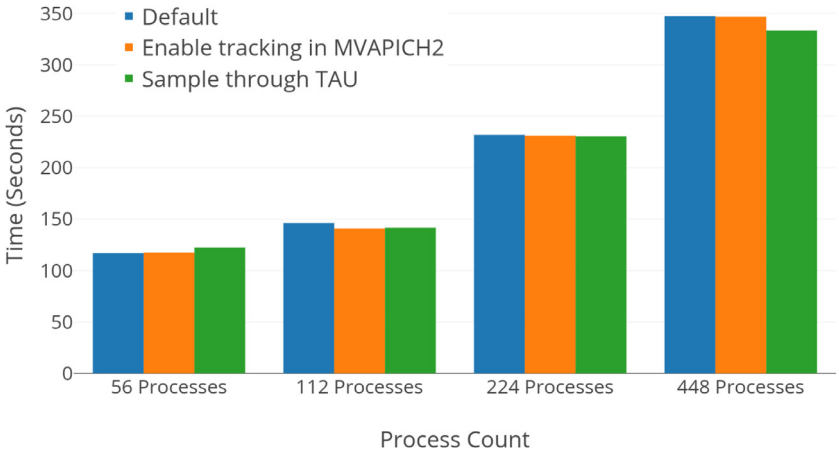


Fig. 9. Overhead in Enabling MPI_T for 3DStencil.

7.2.4. 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.

7.3. 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 Fig. 9.

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 bars with the results because there was negligible variation between runs. Fig. 10 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

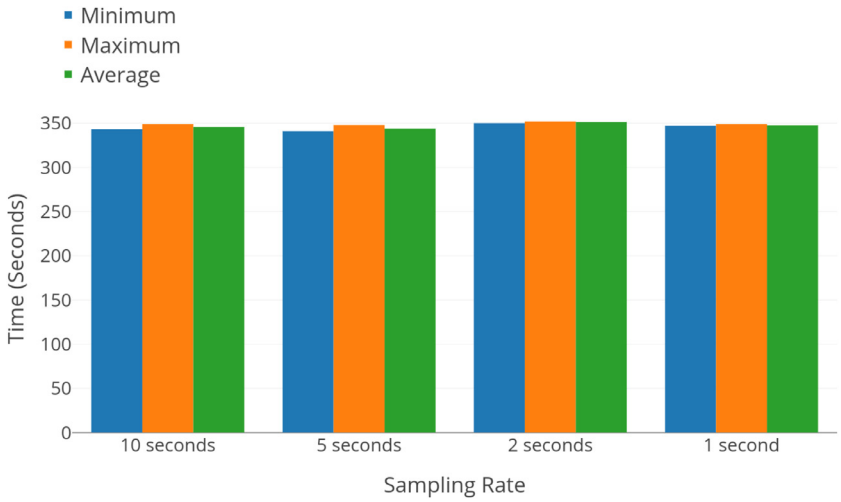


Fig. 10. Effect of MPI_T sampling frequency on overhead for 3DStencil at 448 processes.

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.

8. Discussion

MPI_T allows a performance profiler such as TAU to play a more active role in MPI performance engineering. As we have demonstrated with experiments 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.

By integrating BEACON and PYCOOLR into our infrastructure, we offer the capability of online monitoring to users who may be interested in tracking performance variables while the application is running. The user can also issue CVAR updates through the PYCOOLR GUI and monitor the impact of modifying CVARs. We envision this scenario to be part of an experimentation phase in the performance engineering process, where the user determines the right settings for an application.

However, it may not always be possible or desirable for the user to directly intervene in the performance tuning process. For example, during the process of freeing unused VBUFs, different processes may display different levels of VBUF usage and will require different values for the CVAR `MPIR_CVAR_VBUF_POOL_REDUCED_VALUE`. In such a situation, a more viable option is to have an external software perform the tuning based on a user-defined policy. The plugin infrastructure support inside TAU offers a clean design for developing customized performance tuning policies through MPI_T. This is one step toward enabling complete autotuning of MPI applications. Expert knowledge gathered through monitoring and manual tuning enabled by BEACON can eventually become part of the autotuning and recommendation engines enabled by the plugin infrastructure.

One of the challenges we want to address in the future lies in developing autotuning policies to cover a wider class of applications — multi-phased applications in the climate modeling, computational fluid dynamics, and geophysics domains. Accordingly, we would need to enrich MVAPICH2 to support additional PVARs and CVARs of interest. Providing the user with a more interactive performance engineering functionality is another area we would like to further explore.

9. Conclusion and future work

This paper presented an infrastructure dedicated to MPI Performance Engineering, enabling introspection of MPI runtimes. To serve that purpose, our infrastructure utilized the MPI Tools Information Interface, introduced in the MPI 3.0 standard. We gathered existing components — namely the TAU Performance System and MVAPICH2 and extended them to fully exploit features offered by MPI_T. We 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.

We plan to enrich our infrastructure by 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.

Acknowledgment

The authors would like to thank Jerome Vienne at the Texas Advanced Computing Center for providing access to the AmberMD [8] software for our experiments.

This work was supported by the NSF under the ACI-1450440 & ACI-1450471 grants. This work used the Extreme Science and Discovery Environment (XSEDE) which is supported by National Science Foundation grant number ACI-1053575. This work used allocation grants TG-ASC090010 & TG-NCR130002.

The ri2 cluster at The Ohio State University is supported by the NSF under the NSF-CNS-1513120 grant.

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. http://www.tacc.utexas.edu.

References

[1] M. Forum, MPI: A Message-Passing Interface Standard. Version 3.1, 2015, http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. (June. 2015).
[2] E. Karrels, E. Lusk, Performance analysis of MPI programs, Environ. Tools Parallel Sci. Comput. (1994) 195–200.
[3] S.S. Shende, A.D. Malony, The TAU Parallel Performance System, Int. J. High Perform. Comput. Appl. 20 (2) (2006) 287–311, doi:10.1177/1094342006064482. http://tau.uoregon.edu.
[4] J. Liu, J. Wu, S.P. Kini, P. Wyckoff, D.K. Panda, High performance RDMA-based MPI implementation over InfiniBand, in: Proceedings of the 17th Annual International Conference on Supercomputing, ACM, 2003, pp. 295–304. http://mvapich.cse.ohio-state.edu/.
[5] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. 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, Springer, 2004, pp. 97–104.
[6] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel comput. 22 (6) (1996) 789–828.
[7] M. Pérache, H. Jourdren, R. Namyst, MPC: A Unified Parallel Runtime for Clusters of NUMA Machines, in: Proceedings of the 14th International Euro-Par Conference on Parallel Processing, in: Euro-Par08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 78–88, doi:10.1007/978-3-540-85451-7_9.
[8] D.A. Case, T.E. Cheatham, T. Darden, H. Gohlke, R. Luo, K.M. Merz, A. Onufriev, C. Simmerling, B. Wang, R.J. Woods, The Amber biomolecular simulation programs, J. Comput. Chem. 26 (16) (2005) 1668–1688. http://ambermd.org/.
[9] M.A. Heroux, D.W. Doerfler, P.S. Crozier, J.M. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E.R. Keiter, H.K. Thornquist, R.W. Numrich, Improving Performance via Mini-Applications, Tech. Rep. SAND2009-5574, Sandia National Laboratories, 2009. https://mantevo.org/.
[10] R. Keller, G. Bosilca, G. Fagg, M. Resch, J.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, Springer-Verlag, Bonn, Germany, 2006.
[11] T. Islam, K. Mohror, M. Schulz, Exploring the Capabilities of the New MPI_T Interface, in: Proceedings of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, ACM, New York, NY, USA, 2014, pp. 91:91–91:96, doi:10.1145/2642769.2642781. https://computation.llnl.gov/projects/mpi_t/gyan.
[12] E. Gallardo, J. Vienne, L. Fialho, P. Teller, J. Browne, MPI Advisor: a minimal overhead tool for MPI library performance tuning, in: Proceedings of the 22Nd European MPI Users' Group Meeting, EuroMPI '15, ACM, New York, NY, USA, 2015, pp. 6:1–6:10, doi:10.1145/2802658.2802667.
[13] E. Gallardo, J. Vienne, L. Fialho, P. Teller, J. Browne, Employing MPI_T in MPI Advisor to optimize application performance, Int. J. High Perf. Comput. Appl.0 (0)(0),1094342016684005. doi:10.1177/1094342016684005
[14] J. Vetter, C. Chambreau, mpiP: Lightweight, scalable mpi profiling(2005). http://mpip.sourceforge.net.
[15] M. Chaarawi, J.M. Squyres, E. Gabriel, S. Feki, A Tool for Optimizing Runtime Parameters of Open MPI, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 210–217, doi:10.1007/978-3-540-87475-1_30. https://www.open-mpi.org/projects/otpo/.
[16] M. Gerndt, M. Ott, Automatic performance analysis with periscope, Concurr. Comput. 22 (6) (2010) 736–748, doi:10.1002/cpe.v22:6. http://periscope.in.tum.de/.
[17] A. Sikora, E. César, I. Comprés, M. 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, ACM, New York, NY, USA, 2016, pp. 31–38, doi:10.1145/2916026.2916028.
[18] S. Pellegrini, T. Fahringer, H. Jordan, H. Moritsch, Automatic tuning of MPI runtime parameter settings by using machine learning, in: Proceedings of the 7th ACM International Conference on Computing Frontiers, in: CF '10, ACM, New York, NY, USA, 2010, pp. 115–116, doi:10.1145/1787275.1787310.
[19] K. Huck, S. Shende, A. Malony, H. Kaiser, A. Porterfield, R. Fowler, R. 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, in: ROSS'13, ACM, New York, NY, USA, 2013, pp. 8:1–8:8, doi:10.1145/2491661.2481434. http://khuck.github.io/xpress-apex/.
[20] S. Perarnau, R. Gupta, P. Beckman, et al., Argo: An Exascale Operating System and Runtime, 2015a, http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/poster_files/post298s2-file2.pdf.
[21] S. Perarnau, R. Thakur, K. Iskra, K. Raffenetti, F. Cappello, R. Gupta, P. Beckman, M. Snir, H. Hoffmann, M. Schulz, B. 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, Springer-Verlag New York, Inc., New York, NY, USA, 2015, pp. 173–178, doi:10.1007/978-3-319-19129-4_14.
[22] R.J. Zerr, R.S. Baker, SNAP: SN (discrete ordinates) application proxy: Description, Tech. Rep. LAUR-13-21070, Los Alamos National Laboratories, 2013. https://github.com/lanl/SNAP/.
[23] R.E. Alcouffe, R.S. Baker, J.A. Dahl, S.A. Turner, R. Ward, PARTISN: A Time-Dependent, Parallel Neutral Particle Transport Code System, Los Alamos National Laboratory, 2005. LA-UR-05-3925 (May 2005).
[24] L. Livermore, The Accelerated Strategic Computing Initiative (ASCI) Sweep3d Benchmark Code, Los Alamos, and Sandia National Laboratories., 1995.
[25] K.A. Lindlan, J. Cuny, A.D. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen, A tool framework for static and dynamic analysis of object-oriented software with templates, in: Supercomputing, ACM/IEEE 2000 Conference, IEEE, 2000, p. 49.
[26] R.L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. 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, IEEE Press, 2016, pp. 1–10.
[27] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M.S. Müller, W.E. Nagel, The vampir performance analysis tool-set, in: Tools for High Performance Computing, Springer, 2008, pp. 139–155. http://www.vampir.eu.
[28] TACC Stampede cluster, The University of Texas at Austin: http://www.tacc.utexas.edu.